

.NET fundamentals documentation

Learn the fundamentals of .NET, an open-source developer platform for building many different types of applications.

Learn about .NET

DOWNLOAD

[Download .NET ↗](#)

OVERVIEW

[What is .NET? ↗](#)

[Introduction to .NET](#)

[.NET languages](#)

CONCEPT

[.NET Standard](#)

[Common Language Runtime \(CLR\)](#)

[.NET Core support policy ↗](#)

WHAT'S NEW

[What's new in .NET 8](#)

[What's new in .NET 7](#)

[What's new in .NET 6](#)

[What's new in .NET 5](#)

[What's new in .NET Core 3.1](#)

[What's new in .NET Core 3.0](#)

Install .NET

OVERVIEW

Select which .NET version to use

HOW-TO GUIDE

[Install .NET SDK](#)

[Install .NET runtime](#)

[Use a Linux package manager to install](#)

[Check installed versions](#)

REFERENCE

[.NET SDK and runtime dependencies](#)

Get started with .NET

GET STARTED

[Get started with .NET](#)

[Get started with ASP.NET Core](#)

[.NET on Q&A](#)

[.NET tech community forums ↗](#)

VIDEO

[Tutorial: Hello World in 10 minutes ↗](#)

TUTORIAL

[Create a Hello World app in Visual Studio Code](#)

[Create a Hello World app in Visual Studio](#)

[Containerize a .NET Core app](#)

CONCEPT

[Port from .NET Framework to .NET](#)

DEPLOY

[App publishing](#)

[Publish .NET apps with GitHub Actions](#)

Serialize data

CONCEPT

[Serialize and deserialize JSON](#)

HOW-TO GUIDE

[Serialize and deserialize JSON using C#](#)

[Migrate from Newtonsoft.Json to System.Text.Json](#)

[Write custom converters for JSON serialization](#)

SAMPLE

[Examples of XML serialization](#)

Runtime libraries

OVERVIEW

[Runtime libraries overview](#)

CONCEPT

[Dependency injection in .NET](#)

[Configuration in .NET](#)

[Logging in .NET](#)

[.NET generic host](#)

[Worker services in .NET](#)

[Caching in .NET](#)

[HTTP in .NET](#)

[Localization in .NET](#)

File globbing in .NET

TUTORIAL

[Implement a custom configuration provider](#)

[Compile-time logging source generation](#)

[Create a Windows Service using BackgroundService](#)

Format and convert dates, numbers, and strings

CONCEPT

[Numeric format strings](#)

[Date and time format strings](#)

[Composite formatting](#)

[Convert times between time zones](#)

[Trim and remove characters from strings](#)

[Regular expressions in .NET](#)

HOW-TO GUIDE

[Convert strings to DateTime](#)

[Pad a number with leading zeros](#)

[Display milliseconds in date and time values](#)

REFERENCE

[Regular expression language](#)

Use events and exceptions

CONCEPT

[Best practices for exceptions](#)

[Handle and raise events](#)

 **HOW-TO GUIDE**

[Use a try-catch block to catch exceptions](#)

[Raise and consume events](#)

File and stream I/O

 **CONCEPT**

[File and stream I/O](#)

[File path formats on Windows](#)

 **HOW-TO GUIDE**

[Write text to a file](#)

[Read text from a file](#)

[Compress and extract files](#)

[Open and append to a log file](#)

Get started with .NET

Article • 08/12/2022

This article teaches you how to create and run a "Hello World!" app with [.NET](#).

Create an application

First, download and install the [.NET SDK](#) on your computer.

Next, open a terminal such as **PowerShell**, **Command Prompt**, or **bash**.

Type the following commands:

.NET CLI

```
dotnet new console -o sample1  
cd sample1  
dotnet run
```

You should see the following output:

Output

```
Hello World!
```

Congratulations! You've created a simple .NET application.

Next steps

Get started on developing .NET applications by following a [step-by-step tutorial](#) or by watching [.NET 101 videos](#) on YouTube.

 Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

more information, see [our contributor guide](#).

Tutorials for getting started with .NET

Article • 09/08/2023

The following step-by-step tutorials run on Windows, Linux, or macOS, except as noted.

Tutorials for creating apps

- Create a console app
 - [using Visual Studio Code](#)
 - [using Visual Studio \(Windows\)](#)
- Create a web app
 - [with server-side web UI](#)
 - [with client-side web UI ↗](#)
- Create a web API
- Create a remote procedure call web app
- Create a real-time web app
- Create a serverless function in the cloud
- [Create a mobile app for Android and iOS ↗](#) (Windows)
- Create a Windows desktop app
 - [WPF](#)
 - [Windows Forms](#)
 - [Universal Windows Platform \(UWP\)](#)
- [Create a game using Unity ↗](#)
- Create a Windows service

Tutorials for creating class libraries

- Create a class library
 - [using Visual Studio Code](#)
 - [using Visual Studio \(Windows\)](#)

Resources for learning .NET languages

- [Get started with C#](#)
- [Get started with F#](#)
- [Get started with Visual Basic](#)

Other get-started resources

The following resources are for getting started with developing .NET apps but aren't step-by-step tutorials:

- [Internet of Things \(IoT\)](#)
- [Machine learning](#)

Next steps

To learn more about .NET, see [Introduction to .NET](#).

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

Install .NET on Windows, Linux, and macOS

Learn about installing .NET on Windows, Linux, and macOS. Discover the dependencies required to develop, deploy, and run .NET apps.

Windows



OVERVIEW

[Install on Windows](#)

[Supported Windows releases](#)

[Dependencies](#)

[Install with Visual Studio](#)

[Install alongside Visual Studio Code](#)

macOS



OVERVIEW

[Install on macOS](#)

[Supported macOS releases](#)

[Install alongside Visual Studio Code](#)

Linux



OVERVIEW

[Linux overview](#)

[Alpine](#)

[CentOS](#)

[Debian](#)

[Fedora](#)

[openSUSE](#)

[Red Hat Enterprise Linux and CentOS Stream](#)

[SUSE Linux Enterprise Server](#)

[Ubuntu](#)

Q&A

GET STARTED

[Standalone installers](#)

[Visual Studio installers](#)

[Linux feeds](#)

[Docker](#)

[Enterprise deployment](#)

Install .NET on Windows

Article • 12/19/2023

In this article, you learn how to install .NET on Windows. .NET is made up of the runtime and the SDK. The runtime is used to run a .NET app and might be included with the app. The SDK is used to create .NET apps and libraries. The .NET runtime is always installed with the SDK.

The latest version of .NET is 8.0.

[Download .NET](#)

There are two types of supported releases: Long Term Support (LTS) releases and Standard Term Support (STS) releases. The quality of all releases is the same. The only difference is the length of support. LTS releases get free support and patches for three years. STS releases get free support and patches for 18 months. For more information, see [.NET Support Policy](#).

The following table lists the support status of each version of .NET (and .NET Core):

 Expand table

| ✓ Supported | ✗ Unsupported |
|-------------|---------------|
| 8 (LTS) | 5 |
| 7 (STS) | 3.1 |
| 6 (LTS) | 3.0 |
| | 2.1 |
| | 2.0 |
| | 1.1 |
| | 1.0 |

Install with Windows Package Manager (winget)

You can install and manage .NET through the Windows Package Manager service, using the `winget` tool. For more information about how to install and use `winget`, see [Use the Windows Package Manager](#).

winget tool.

If you're installing .NET system-wide, install with administrative privileges.

Install the SDK

The .NET SDK allows you to develop apps with .NET. If you install the .NET SDK, you don't need to install the corresponding runtimes. To install the .NET SDK, run the following command:

Windows Command Prompt

```
winget install Microsoft.DotNet.SDK.8
```

Install the runtime

There are three different .NET runtimes you can install, however, you should install both the .NET Desktop Runtime and the ASP.NET Core Runtime for maximum compatibility with all types of .NET apps. The following table describes what is included with each runtime:

[+] Expand table

| | Includes .NET Runtime | Includes .NET Desktop Runtime | Includes ASP.NET Core Runtime |
|-----------------------------|------------------------------|--------------------------------------|--------------------------------------|
| .NET Runtime | Yes | No | No |
| .NET Desktop Runtime | Yes | Yes | No |
| ASP.NET Core Runtime | No | No | Yes |

The following list provides details about each runtime along with the winget commands to install them:

- .NET Desktop Runtime

This runtime supports Windows Presentation Foundation (WPF) and Windows Forms apps that are built with .NET. This isn't the same as .NET Framework, which comes with Windows. This runtime includes .NET Runtime, but doesn't include ASP.NET Core Runtime, which must be installed separately.

```
Windows Command Prompt
```

```
winget install Microsoft.DotNet.DesktopRuntime.8
```

- .NET Runtime

This is the base runtime, and contains just the components needed to run a console app. Typically, you'd install both .NET Desktop Runtime and ASP.NET Core Runtime instead of this one.

```
Windows Command Prompt
```

```
winget install Microsoft.DotNet.Runtime.8
```

- ASP.NET Core Runtime

This runtime runs web server apps and provides many web-related APIs. ASP.NET Core Runtime allows you to run apps that were made with .NET that didn't provide the runtime. You must install .NET Runtime in addition to this runtime. The following commands install ASP.NET Core Runtime, In your terminal, run the following commands:

```
Windows Command Prompt
```

```
winget install Microsoft.DotNet.AspNetCore.8
```

You can install preview versions of the runtimes by substituting the version number, such as `6`, with the word `Preview`. The following example installs the preview release of the .NET Desktop Runtime:

```
Windows Command Prompt
```

```
winget install Microsoft.DotNet.DesktopRuntime.Preview
```

Install alongside Visual Studio Code

Visual Studio Code is a powerful and lightweight source code editor that runs on your desktop. Visual Studio Code is available for Windows, macOS, and Linux.

While Visual Studio Code doesn't come with an automated .NET Core installer like Visual Studio does, adding .NET Core support is simple.

1. [Download and install Visual Studio Code](#).
2. [Download and install the .NET SDK](#).
3. [Install the C# extension from the Visual Studio Code marketplace](#).

The **C# For Visual Studio Code** extension includes the latest .NET SDK, and you don't need to install any .NET runtime separately.

Install with Windows Installer

There are three different .NET runtimes you can install, however, you should install both the .NET Desktop Runtime and the ASP.NET Core Runtime for maximum compatibility with all types of .NET apps. The following table describes what is included with each runtime:

[\[+\] Expand table](#)

| | Includes .NET Runtime | Includes .NET Desktop Runtime | Includes ASP.NET Core Runtime |
|-----------------------------|------------------------------|--------------------------------------|--------------------------------------|
| .NET Runtime | Yes | No | No |
| .NET Desktop Runtime | Yes | Yes | No |
| ASP.NET Core Runtime | No | No | Yes |

.NET SDK allows you to create .NET apps, and includes all of the runtimes.

The [download page](#) for .NET provides Windows Installer executables.

If you want to install .NET silently, such as in a production environment or to support continuous integration, use the following switches:

- `/install`
Installs .NET.
- `/quiet`
Prevents any UI and prompts from displaying.
- `/norestart`
Suppresses any attempts to restart.

```
dotnet-sdk-8.0.100-win-x64.exe /install /quiet /norestart
```

For more information, see [Standard Installer Command-Line Options](#).

Tip

The installer returns an exit code of 0 for success and an exit code of 3010 to indicate that a restart is required. Any other value is generally an error code.

Install with PowerShell automation

The [dotnet-install scripts](#) are used for CI automation and non-admin installs of the runtime. You can download the script from the [dotnet-install script reference page](#).

The script defaults to installing the latest [long term support \(LTS\)](#) version, which is .NET 8. You can choose a specific release by specifying the `-Channel` switch. Include the `-Runtime` switch to install a runtime. Otherwise, the script installs the SDK.

The following command installs both the Desktop and ASP.NET Core runtimes for maximum compatibility.

PowerShell

```
dotnet-install.ps1 -Channel 8.0 -Runtime windowsdesktop  
dotnet-install.ps1 -Channel 8.0 -Runtime aspnetcore
```

Install the SDK by omitting the `-Runtime` switch. The `-Channel` switch is set in this example to `STS`, which installs the latest Standard Term Support version, which is .NET 7.

PowerShell

```
dotnet-install.ps1 -Channel STS
```

Install with Visual Studio

If you're using Visual Studio to develop .NET apps, the following table describes the minimum required version of Visual Studio based on the target .NET SDK version.

[+] Expand table

| .NET SDK version | Visual Studio version |
|-------------------------|--|
| 8 | Visual Studio 2022 version 17.8 or higher. |
| 7 | Visual Studio 2022 version 17.4 or higher. |
| 6 | Visual Studio 2022 version 17.0 or higher. |
| 5 | Visual Studio 2019 version 16.8 or higher. |
| 3.1 | Visual Studio 2019 version 16.4 or higher. |
| 3.0 | Visual Studio 2019 version 16.3 or higher. |
| 2.2 | Visual Studio 2017 version 15.9 or higher. |
| 2.1 | Visual Studio 2017 version 15.7 or higher. |

If you already have Visual Studio installed, you can check your version with the following steps.

1. Open Visual Studio.
2. Select **Help > About Microsoft Visual Studio**.
3. Read the version number from the **About** dialog.

Visual Studio can install the latest .NET SDK and runtime.

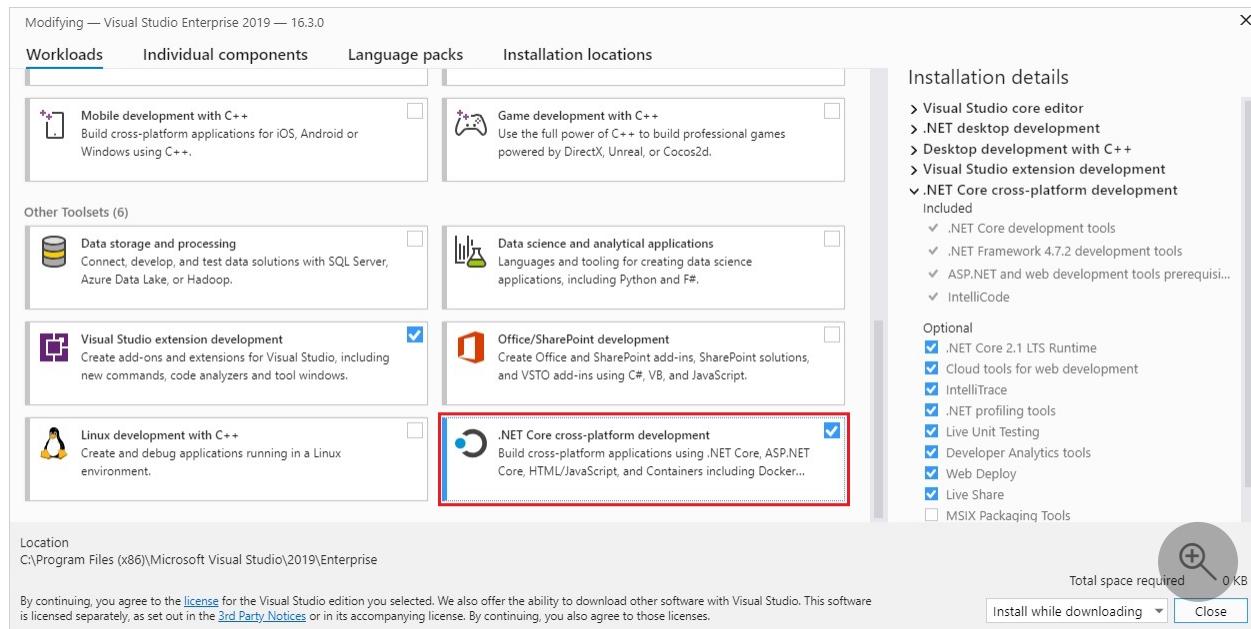
[Download Visual Studio](#)

For more information, see [.NET SDK, MSBuild, and Visual Studio versioning](#).

Select a workload

When installing or modifying Visual Studio, select one or more of the following workloads, depending on the kind of application you're building:

- The **.NET Core cross-platform development** workload in the **Other Toolsets** section.
- The **ASP.NET and web development** workload in the **Web & Cloud** section.
- The **Azure development** workload in the **Web & Cloud** section.
- The **.NET desktop development** workload in the **Desktop & Mobile** section.



Supported releases

The following table is a list of currently supported .NET releases and the versions of Windows they're supported on. These versions remain supported until either the version of [.NET reaches end-of-support](#) or the version of [Windows reaches end-of-life](#).

Windows 10 versions end-of-service dates are segmented by edition. Only **Home**, **Pro**, **Pro Education**, and **Pro for Workstations** editions are considered in the following table. Check the [Windows lifecycle fact sheet](#) for specific details.

Tip

A + symbol represents the minimum version.

[Expand table](#)

| Operating System | .NET 8 | .NET 7 | .NET 6 |
|---------------------------------------|--------|--------|--------|
| Windows 11 | ✓ | ✓ | ✓ |
| Windows Server 2022 | ✓ | ✓ | ✓ |
| Windows Server, Version 1903 or later | ✓ | ✓ | ✓ |
| Windows 10, Version 1607 or later | ✓ | ✓ | ✓ |
| Windows 8.1 | ✗ | ✗ | ✓ |
| Windows 7 SP1 ESU | ✗ | ✗ | ✓ |

| Operating System | .NET 8 | .NET 7 | .NET 6 |
|-----------------------------|--------|--------|--------|
| Windows Server 2019 | ✓ | ✓ | ✓ |
| Windows Server 2016 | | | |
| Windows Server 2012 R2 | | | |
| Windows Server 2012 | | | |
| Windows Server Core 2012 R2 | ✓ | ✓ | ✓ |
| Windows Server Core 2012 | ✓ | ✓ | ✓ |
| Nano Server, Version 1809+ | ✓ | ✓ | ✓ |
| Nano Server, Version 1803 | ✗ | ✗ | ✗ |

For more information about .NET 8 supported operating systems, distributions, and lifecycle policy, see [.NET 8 Supported OS Versions](#).

Unsupported releases

The following versions of .NET are ✗ no longer supported:

- .NET 5
- .NET Core 3.1
- .NET Core 3.0
- .NET Core 2.2
- .NET Core 2.1
- .NET Core 2.0

Verify downloaded binaries

After downloading an installer, verify it to make sure that the file hasn't been changed or corrupted. You can verify the checksum on your computer and then compare it to what was reported on the download website.

When you download an installer or binary from an official download page, the checksum for the file is displayed. Select the **Copy** button to copy the checksum value to your clipboard.

Thanks for downloading .NET 8.0 SDK (v8.0.100) - Windows x64 Installer!

 **Using Visual Studio?** This release is only compatible with Visual Studio 2022 (v17.8). Using a different version? See [.NET SDKs for Visual Studio](#).

If your download doesn't start after 30 seconds, [click here to download manually](#).

| | | |
|-------------------|---|---|
| Direct link | https://download.visualstudio.microsoft.com/download/pr/93961dfb-d1e0-49c8-9230-abcba1ebab5a/811ed1eb63d7652325727720edda26a8/dotnet-sdk-8.0.100-win-x64.exe | <input type="button" value="Copy"/> |
| Checksum (SHA512) | 248acec95b381e5302255310fb9396267fd74a4a2dc2c3a5989031969cb31f8270cbd14bda1b0352ac90f8138bddad1a58e4af1e56cc4a1613b1cf2854b518e | <input style="border: 2px solid red; border-radius: 5px; padding: 2px 10px;" type="button" value="Copy"/> |

You can use **PowerShell** or **Command Prompt** to validate the checksum of the file you've downloaded. For example, the following command reports the checksum of the *dotnet-sdk-8.0.100-win-x64.exe* file:

Windows Command Prompt

```
> certutil -hashfile dotnet-sdk-8.0.100-win-x64.exe SHA512
SHA512 hash of dotnet-sdk-8.0.100-win-x64.exe:
248acec95b381e5302255310fb9396267fd74a4a2dc2c3a5989031969cb31f8270cbd14bda1b
c0352ac90f8138bddad1a58e4af1e56cc4a1613b1cf2854b518e
CertUtil: -hashfile command completed successfully.
```

PowerShell

```
> (Get-FileHash .\dotnet-sdk-8.0.100-win-x64.exe -Algorithm SHA512).Hash
248acec95b381e5302255310fb9396267fd74a4a2dc2c3a5989031969cb31f8270cbd14bda1b
c0352ac90f8138bddad1a58e4af1e56cc4a1613b1cf2854b518e
```

Compare the checksum with the value provided by the download site.

Use PowerShell and a checksum file to validate

The .NET release notes contain a link to a checksum file you can use to validate your downloaded file. The following steps describe how to download the checksum file and validate a .NET install binary:

1. The release notes page for .NET 8 on GitHub at <https://github.com/dotnet/core/tree/main/release-notes/8.0> contains a section named **Releases**. The table in that section links to the downloads and checksum files for each .NET 8 release:

Releases

| Date | Release |
|------------|---------------------------------|
| 2023/11/14 | 8.0.0 |
| 2023/10/10 | 8.0.0 RC 2 |
| 2023/09/12 | 8.0.0 RC 1 |
| 2023/08/08 | 8.0.0 Preview 7 |
| 2023/07/11 | 8.0.0 Preview 6 |

2. Select the link for the version of .NET that you downloaded. The previous section used .NET SDK 8.0.100, which is in the .NET 8.0.0 release.

Tip

If you're not sure which .NET release contains your checksum file, explore the links until you find it.

3. In the release page, you can see the .NET Runtime and .NET SDK version, and a link to the checksum file:

.NET 8.0.0 - November 14, 2023

The [.NET 8.0.0 and .NET SDK 8.0.100](#) releases are available for download. The latest 8.0 release is always listed at [.NET 8.0 Releases](#).

Downloads

| | SDK Installer ¹ | SDK Binaries ¹ | Runtime Installer | Runtime Binaries | ASP.NET Core Runtime | Windows Desktop Runtime |
|---------|--|---|-----------------------------------|--|--|-----------------------------------|
| Windows | x86 x64 Arm64 | x86 x64 Arm64 | x86 x64 Arm64 | x86 x64 Arm64 | x86 x64 Hosting Bundle² | x86 x64 Arm64 |
| macOS | x64 ARM64 | x64 ARM64 | x64 ARM64 | x64 ARM64 | x64 ARM64 | - |
| Linux | Snap and Package Manager | x64 Arm Arm64 Arm32 Alpine x64 Alpine | Packages (x64) | x64 Arm Arm64 Arm32 Alpine Arm64 Alpine x64 Alpine | x64¹ Arm¹ Arm64¹ x64 Alpine | - |
| | Checksums | Checksums | Checksums | Checksums | Checksums | Checksums |

4. Copy the link to the checksum file.

5. Use the following script, but replace the link to download the appropriate checksum file:

```
PowerShell
```

```
Invoke-WebRequest  
https://dotnetcli.blob.core.windows.net/dotnet/checksums/8.0.0-sha.txt  
-OutFile 8.0.0-sha.txt
```

- With both the checksum file and the .NET release file downloaded to the same directory, search the checksum file for the checksum of the .NET download:

When validation passes, you see **True** printed:

PowerShell

```
> (Get-Content .\8.0.0-sha.txt | Select-String "dotnet-sdk-8.0.100-win-x64.exe").Line -like (Get-FileHash .\dotnet-sdk-8.0.100-win-x64.exe -Algorithm SHA512).Hash + "*"  
True
```

If you see **False** printed, the file you downloaded isn't valid and shouldn't be used.

Runtime information

The runtime is used to run apps created with .NET. When an app author publishes an app, they can include the runtime with their app. If they don't include the runtime, it's up to the user to install the runtime.

There are three different .NET runtimes you can install, however, you should install both the .NET Desktop Runtime and the ASP.NET Core Runtime for maximum compatibility with all types of .NET apps. The following table describes what is included with each runtime:

[+] Expand table

| | Includes .NET Runtime | Includes .NET Desktop Runtime | Includes ASP.NET Core Runtime |
|-----------------------------|-----------------------|-------------------------------|-------------------------------|
| .NET Runtime | Yes | No | No |
| .NET Desktop Runtime | Yes | Yes | No |
| ASP.NET Core Runtime | No | No | Yes |

The following list provides details about each runtime:

- *Desktop Runtime*
Runs .NET WPF and Windows Forms desktop apps for Windows. Includes the .NET runtime.
- *ASP.NET Core Runtime*
Runs ASP.NET Core apps.
- *.NET Runtime*
This runtime is the simplest runtime and doesn't include any other runtime. Install both *ASP.NET Core Runtime* and *Desktop Runtime* for the best compatibility with .NET apps.

[Download .NET Runtime](#)

SDK information

The SDK is used to build and publish .NET apps and libraries. Installing the SDK includes all three [runtimes](#): ASP.NET Core, Desktop, and .NET.

[Download .NET SDK](#)

Arm-based Windows PCs

The following sections describe things you should consider when installing .NET on an Arm-based Windows PC.

What is supported

The following table describes which versions of .NET are supported on an Arm-based Windows PC:

[\[+\] Expand table](#)

| .NET Version | Architecture | SDK | Runtime | Path conflict |
|--------------|--------------|-----|---------|---------------|
| 8 | Arm64 | Yes | Yes | No |
| 8 | x64 | Yes | Yes | No |
| 7 | Arm64 | Yes | Yes | No |
| 7 | x64 | Yes | Yes | No |

| .NET Version | Architecture | SDK | Runtime | Path conflict |
|--------------|--------------|-----|---------|---------------|
| 6 | Arm64 | Yes | Yes | No |
| 6 | x64 | Yes | Yes | No |
| 5 | Arm64 | Yes | Yes | Yes |
| 5 | x64 | No | Yes | Yes |

The x64 and Arm64 versions of the .NET SDK exist independently from each other. If a new version is released, each architecture install needs to be upgraded.

Path differences

On an Arm-based Windows PC, all Arm64 versions of .NET are installed to the normal C:\Program Files\dotnet\ folder. However, the **x64** version of the .NET SDK is installed to the C:\Program Files\dotnet\x64\ folder.

Path conflicts

The **x64** .NET SDK installs to its own directory, as described in the previous section. This allows the Arm64 and x64 versions of the .NET SDK to exist on the same machine.

However, any **x64** SDK prior to 6 isn't supported and installs to the same location as the Arm64 version, the C:\Program Files\dotnet\ folder. If you want to install an unsupported x64 SDK, you must uninstall the Arm64 version first. The opposite is also true, you must uninstall the unsupported x64 SDK to install the Arm64 version.

Path variables

Environment variables that add .NET to system path, such as the `PATH` variable, may need to be changed if you have both the x64 and Arm64 versions of the .NET SDK installed. Additionally, some tools rely on the `DOTNET_ROOT` environment variable, which would also need to be updated to point to the appropriate .NET SDK installation folder.

Dependencies

.NET 8

The following Windows versions are supported with .NET 8:

① Note

A + symbol represents the minimum version.

[+] Expand table

| OS | Version | Architectures |
|---------------------|---------|-----------------|
| Windows 11 | 22000+ | x64, x86, Arm64 |
| Windows 10 Client | 1607+ | x64, x86, Arm64 |
| Windows Server | 2012+ | x64, x86 |
| Windows Server Core | 2012+ | x64, x86 |
| Nano Server | 1809+ | x64 |

For more information about .NET 8 supported operating systems, distributions, and lifecycle policy, see [.NET 8 Supported OS Versions](#).

Windows 7 / 8.1 / Server 2012

More dependencies are required if you're installing the .NET SDK or runtime on the following Windows versions:

[+] Expand table

| Operating System | Prerequisites |
|-----------------------------------|---|
| Windows 7 SP1 ESU | - Microsoft Visual C++ 2015-2019 Redistributable 64-bit / 32-bit - KB3063858 64-bit / 32-bit - Microsoft Root Certificate Authority 2011 (.NET Core 2.1 offline installer only) |
| Windows 8.1 | Microsoft Visual C++ 2015-2019 Redistributable 64-bit / 32-bit |
| Windows Server 2012 | Microsoft Visual C++ 2015-2019 Redistributable 64-bit / 32-bit |
| Windows Server 2012 R2 | Microsoft Visual C++ 2015-2019 Redistributable 64-bit / 32-bit |

The previous requirements are also required if you receive an error related to either of the following dlls:

- *api-ms-win-crt-runtime-l1-1-0.dll*
- *api-ms-win-cor-timezone-l1-1-0.dll*
- *hostfxr.dll*

Docker

Containers provide a lightweight way to isolate your application from the rest of the host system. Containers on the same machine share just the kernel and use resources given to your application.

.NET can run in a Docker container. Official .NET Docker images are published to the Microsoft Container Registry (MCR) and are discoverable at the [Microsoft .NET Docker Hub repository](#). Each repository contains images for different combinations of the .NET (SDK or Runtime) and OS that you can use.

Microsoft provides images that are tailored for specific scenarios. For example, the [ASP.NET Core repository](#) provides images that are built for running ASP.NET Core apps in production.

For more information about using .NET in a Docker container, see [Introduction to .NET and Docker](#) and [Samples](#).

Troubleshooting

After installing the .NET SDK, you may run into problems trying to run .NET CLI commands. This section collects those common problems and provides solutions.

- [No .NET SDK was found](#)

No .NET SDK was found

Most likely you installed both the x86 (32-bit) and x64 (64-bit) versions of the .NET SDK. This is causing a conflict because when you run the `dotnet` command it's resolving to the x86 version when it should resolve to the x64 version. This is usually fixed by adjusting the `%PATH%` variable to resolve the x64 version first.

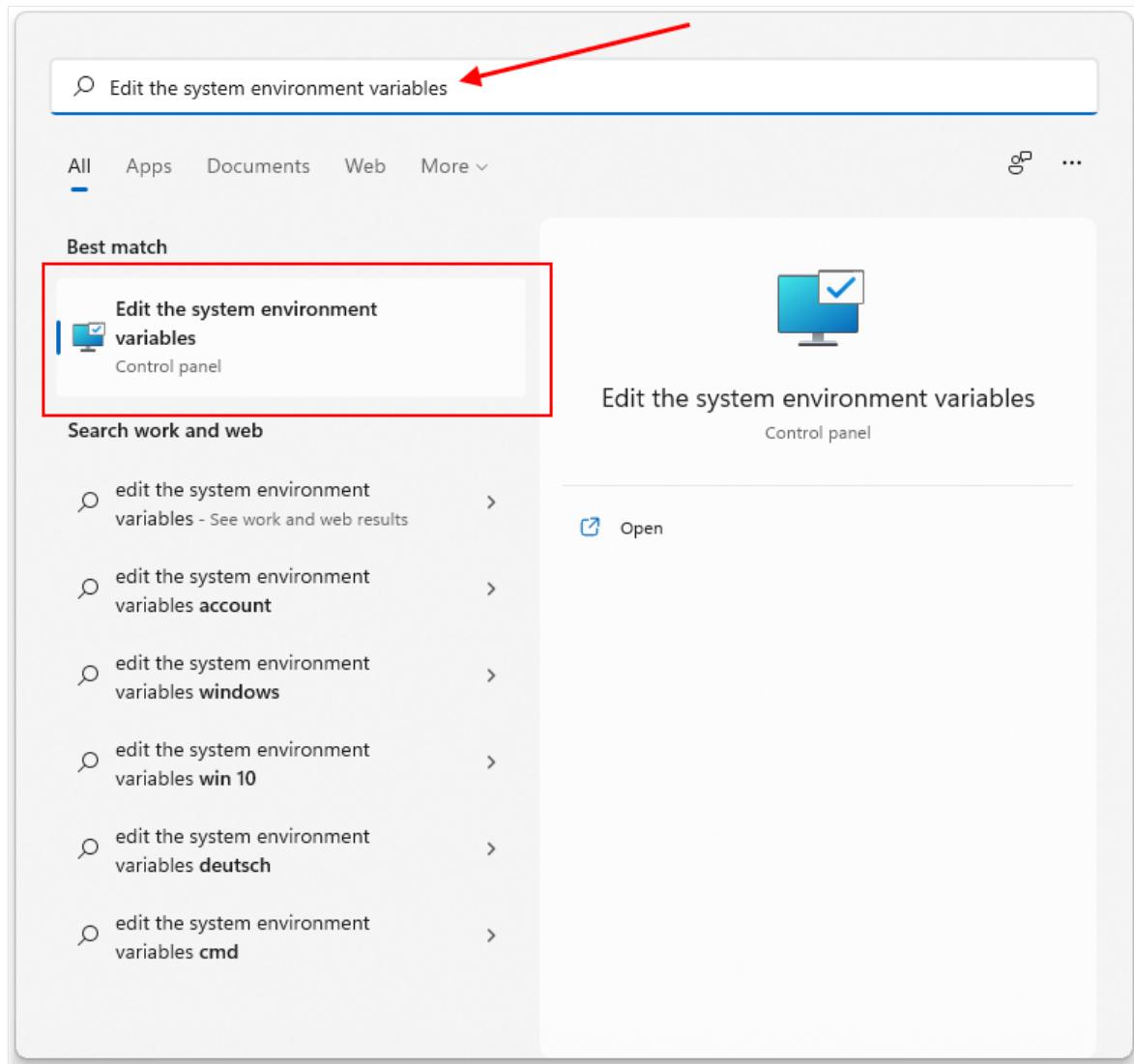
1. Verify that you have both versions installed by running the `where.exe dotnet` command. If you do, you should see an entry for both the *Program Files* and *Program Files (x86)* folders. If the *Program Files (x86)* folder is first, as demonstrated by the following example, it's incorrect and you should continue on to the next step.

Windows Command Prompt

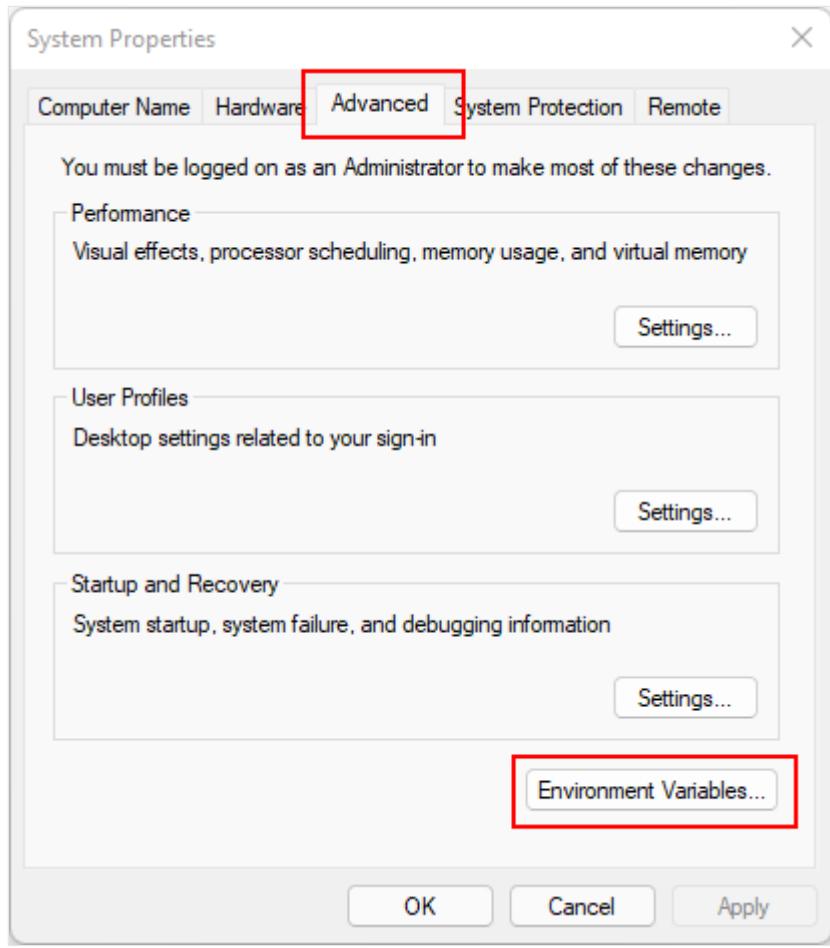
```
> where.exe dotnet
C:\Program Files (x86)\dotnet\dotnet.exe
C:\Program Files\dotnet\dotnet.exe
```

If it's correct and the *Program Files* is first, you don't have the problem this section is discussing and you should create a [.NET help request issue on GitHub](#) ↗

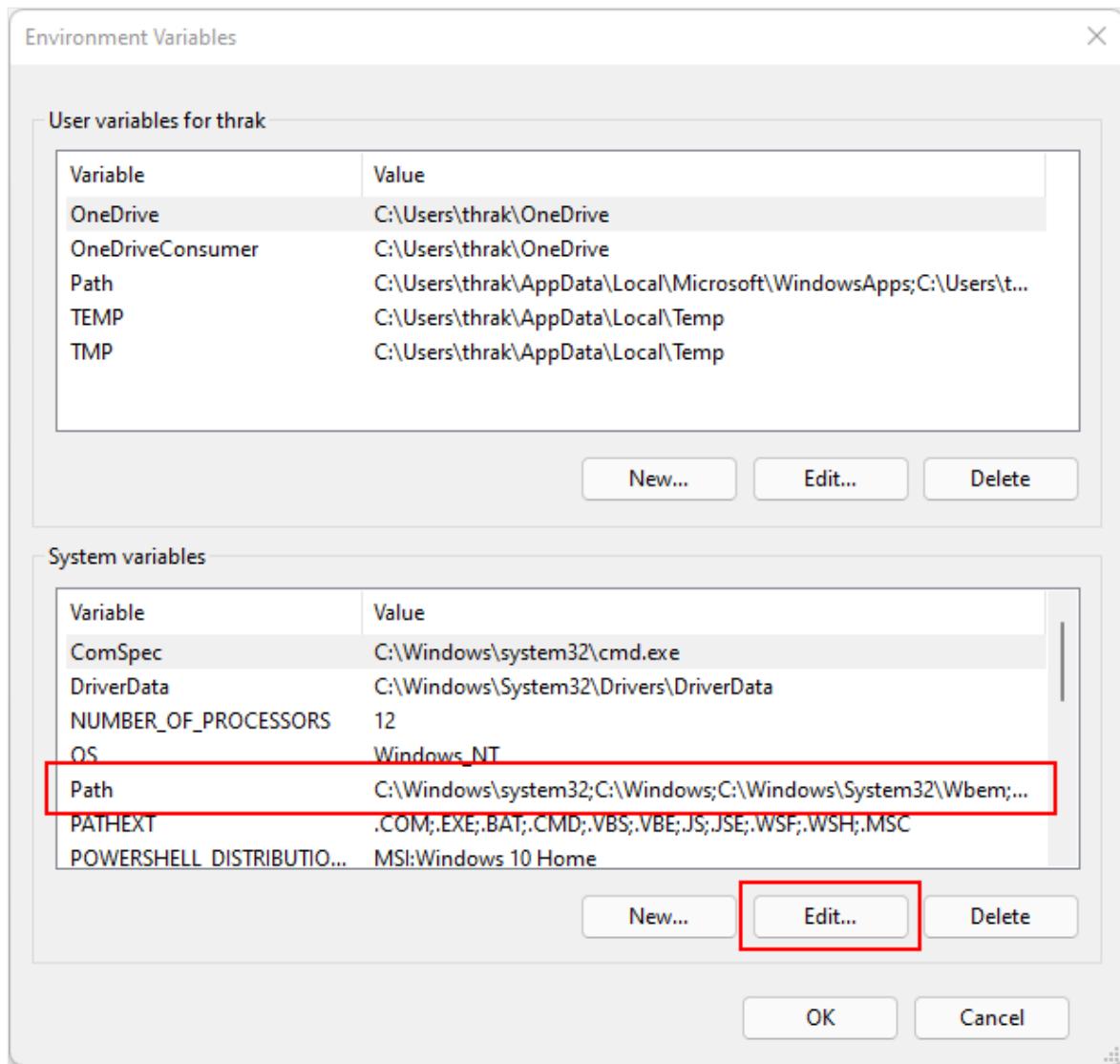
2. Press the Windows button and type "Edit the system environment variables" into search. Select **Edit the system environment variables**.



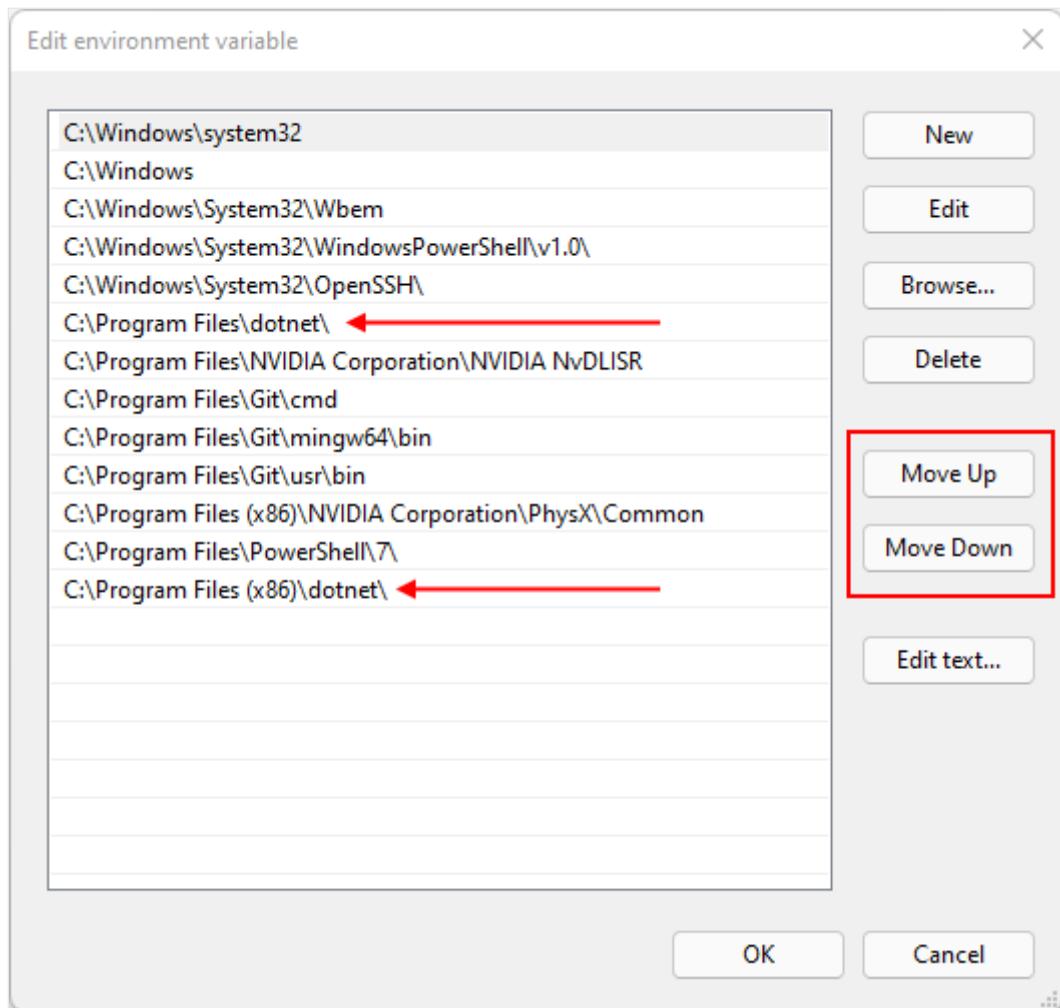
3. The **System Properties** window opens up to the **Advanced Tab**. Select **Environment Variables**.



4. On the **Environment Variables** window, under the **System variables** group, select the *Path** row and then select the **Edit** button.



5. Use the Move Up and Move Down buttons to move the C:\Program Files\dotnet\ entry above C:\Program Files (x86)\dotnet\.



Next steps

- How to check if .NET is already installed.
- Tutorial: Hello World tutorial.
- Tutorial: Create a new app with Visual Studio Code.
- Tutorial: Containerize a .NET Core app.

 Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



.NET feedback

.NET is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

Install .NET on macOS

Article • 12/30/2023

In this article, you learn how to install .NET on macOS. .NET is made up of the runtime and the SDK. The runtime is used to run a .NET app and might or might not be included with the app. The SDK is used to create .NET apps and libraries. The .NET runtime is always installed with the SDK.

The latest version of .NET is 8.

[Download .NET](#)

Supported releases

There are two types of supported releases: Long Term Support (LTS) releases and Standard Term Support (STS) releases. The quality of all releases is the same. The only difference is the length of support. LTS releases get free support and patches for three years. STS releases get free support and patches for 18 months. For more information, see [.NET Support Policy](#).

The following table is a list of currently supported .NET releases and the versions of macOS they're supported on:

[+] Expand table

| Operating System | .NET 8 (LTS) | .NET 7 (STS) | .NET 6 (LTS) |
|------------------------|--------------|--------------|--------------|
| macOS 14.0 "Sonoma" | ✓ 8.0 | ✓ 7.0 | ✓ 6.0 |
| macOS 13.0 "Ventura" | ✓ 8.0 | ✓ 7.0 | ✓ 6.0 |
| macOS 12.0 "Monterey" | ✓ 8.0 | ✓ 7.0 | ✓ 6.0 |
| macOS 11.0 "Big Sur" | ✗ | ✓ 7.0 | ✓ 6.0 |
| macOS 10.15 "Catalina" | ✗ | ✓ 7.0 | ✓ 6.0 |

For a full list of .NET versions and their support life cycle, see [.NET Support Policy](#).

Unsupported releases

The following versions of .NET are  no longer supported:

- .NET 5
- .NET Core 3.1
- .NET Core 3.0
- .NET Core 2.2
- .NET Core 2.1
- .NET Core 2.0

Runtime information

The runtime is used to run apps created with .NET. When an app author publishes an app, they can include the runtime with their app. If they don't include the runtime, it's up to the user to install the runtime.

There are two different runtimes you can install on macOS:

- *ASP.NET Core runtime*
Runs ASP.NET Core apps. Includes the .NET runtime.
- *.NET runtime*
This runtime is the simplest runtime and doesn't include any other runtime. It's highly recommended that you install *ASP.NET Core runtime* for the best compatibility with .NET apps.

[Download .NET Runtime](#)

SDK information

The SDK is used to build and publish .NET apps and libraries. Installing the SDK includes both [runtimes](#): ASP.NET Core and .NET.

Notarization

Beginning with macOS Catalina (version 10.15), all software built after June 1, 2019 that's distributed with Developer ID must be notarized. This requirement applies to the .NET runtime, .NET SDK, and software created with .NET.

The runtime and SDK installers for .NET have been notarized since February 18, 2020. Prior released versions aren't notarized. If you run a non-notarized app, you'll see an error similar to the following image:



For more information about how enforced-notarization affects .NET (and your .NET apps), see [Working with macOS Catalina Notarization](#).

libgdiplus

.NET applications that use the *System.Drawing.Common* assembly require libgdiplus to be installed.

An easy way to obtain libgdiplus is by using the [Homebrew \("brew"\)](#) package manager for macOS. After installing *brew*, install libgdiplus by executing the following commands at a Terminal (command) prompt:

Console

```
brew update  
brew install mono-libgdiplus
```

Automated install

macOS has standalone installers that can be used to install .NET:

- [.NET 8 downloads](#)
- [.NET 7 downloads](#)
- [.NET 6 downloads](#)

Manual install

As an alternative to the macOS installers for .NET, you can download and manually install the SDK and runtime. Manual installation is usually performed as part of continuous integration testing. For a developer or user, it's generally better to use an [installer ↗](#).

Download a **binary** release for either the SDK or the runtime from one of the following sites. The .NET SDK includes the corresponding runtime:

- [✓ .NET 8 downloads ↗](#)
- [✓ .NET 7 downloads ↗](#)
- [✓ .NET 6 downloads ↗](#)
- [All .NET downloads ↗](#)

Extract the downloaded file and use the `export` command to set `DOTNET_ROOT` to the extracted folder's location and then ensure .NET is in PATH. Exporting `DOTNET_ROOT` makes the .NET CLI commands available in the terminal. For more information about .NET environment variables, see [.NET SDK and CLI environment variables](#).

Different versions of .NET can be extracted to the same folder, which coexist side-by-side.

Example

The following commands use Bash to set the environment variable `DOTNET_ROOT` to the current working directory followed by `.dotnet`. That directory is created if it doesn't exist. The `DOTNET_FILE` environment variable is the filename of the .NET binary release you want to install. This file is extracted to the `DOTNET_ROOT` directory. Both the `DOTNET_ROOT` directory and its `tools` subdirectory are added to the `PATH` environment variable.

ⓘ Important

If you run these commands, remember to change the `DOTNET_FILE` value to the name of the .NET binary you downloaded.

Bash

```
DOTNET_FILE=dotnet-sdk-8.0.100-osx-x64.tar.gz
export DOTNET_ROOT=$(pwd)/.dotnet

mkdir -p "$DOTNET_ROOT" && tar zxf "$DOTNET_FILE" -C "$DOTNET_ROOT"
```

```
export PATH=$PATH:$DOTNET_ROOT
```

You can install more than one version of .NET in the same folder.

You can also install .NET to the home directory identified by the `HOME` variable or `~` path:

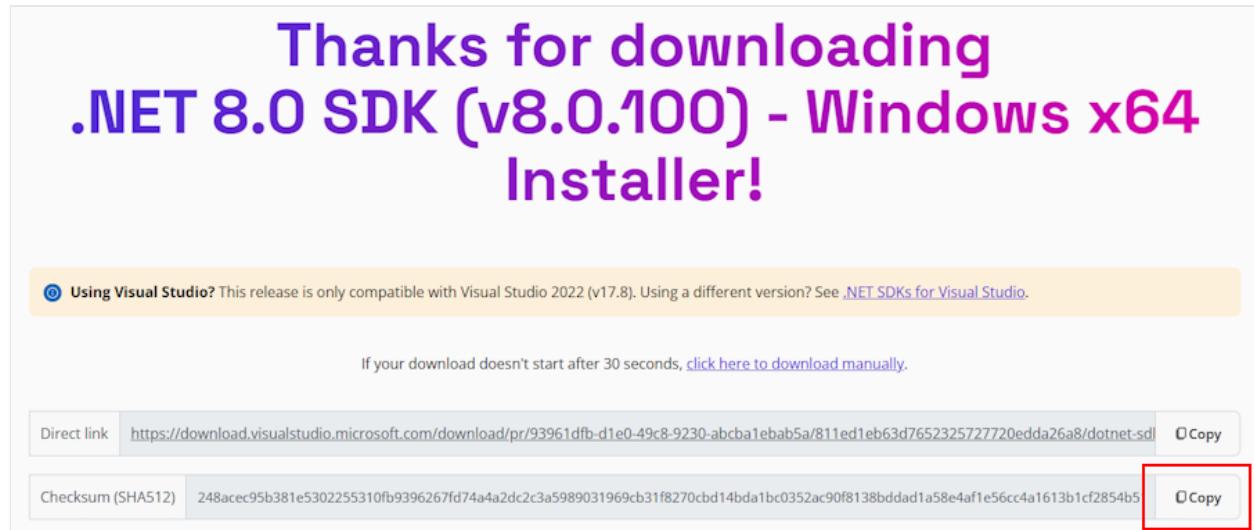
Bash

```
export DOTNET_ROOT=$HOME/.dotnet
```

Verify downloaded binaries

After downloading an installer, verify it to make sure that the file hasn't been changed or corrupted. You can verify the checksum on your computer and then compare it to what was reported on the download website.

When you download an installer or binary from an official download page, the checksum for the file is displayed. Select the **Copy** button to copy the checksum value to your clipboard.



Use the `sha512sum` command to print the checksum of the file you've downloaded. For example, the following command reports the checksum of the `dotnet-sdk-8.0.100-linux-x64.tar.gz` file:

Bash

```
$ sha512sum dotnet-sdk-8.0.100-linux-x64.tar.gz
13905ea20191e70baeba50b0e9bbe5f752a7c34587878ee104744f9fb453bfe439994d389697
22bdae7f60ee047d75dda8636f3ab62659450e9cd4024f38b2a5  dotnet-sdk-8.0.100-
linux-x64.tar.gz
```

Compare the checksum with the value provided by the download site.

ⓘ Important

Even though a Linux file is shown in these examples, this information equally applies to macOS.

Use a checksum file to validate

The .NET release notes contain a link to a checksum file you can use to validate your downloaded file. The following steps describe how to download the checksum file and validate a .NET install binary:

1. The release notes page for .NET 8 on GitHub at

[https://github.com/dotnet/core/tree/main/release-notes/8.0 ↗](https://github.com/dotnet/core/tree/main/release-notes/8.0) contains a section named **Releases**. The table in that section links to the downloads and checksum files for each .NET 8 release:

| Date | Release |
|------------|---------------------------------|
| 2023/11/14 | 8.0.0 |
| 2023/10/10 | 8.0.0 RC 2 |
| 2023/09/12 | 8.0.0 RC 1 |
| 2023/08/08 | 8.0.0 Preview 7 |
| 2023/07/11 | 8.0.0 Preview 6 |

2. Select the link for the version of .NET that you downloaded. The previous section used .NET SDK 8.0.100, which is in the .NET 8.0.0 release.
3. In the release page, you can see the .NET Runtime and .NET SDK version, and a link to the checksum file:

.NET 8.0.0 - November 14, 2023

The .NET 8.0.0 and .NET SDK 8.0.100 releases are available for download. The latest 8.0 release is always listed at [.NET 8.0 Releases](#).

Downloads

| | SDK Installer ¹ | SDK Binaries ¹ | Runtime Installer | Runtime Binaries | ASP.NET Core Runtime | Windows Desktop Runtime |
|---------|--|---|-----------------------------------|--|--|-----------------------------------|
| Windows | x86 x64 Arm64 | x86 x64 Arm64 | x86 x64 Arm64 | x86 x64 Arm64 | x86 x64 Hosting Bundle² | x86 x64 Arm64 |
| macOS | x64 ARM64 | x64 ARM64 | x64 ARM64 | x64 ARM64 | x64 ARM64 | - |
| Linux | Snap and Package Manager | x64 Arm Arm64 Arm32 Alpine x64 Alpine | Packages (x64) | x64 Arm Arm64 Arm32 Alpine Arm64 Alpine x64 Alpine | x64¹ Arm¹ Arm64¹ x64 Alpine | - |
| | Checksums | Checksums | Checksums | Checksums | Checksums | Checksums |

4. Copy the link to the checksum file.

5. Use the following script, but replace the link to download the appropriate checksum file:

Bash

```
curl -O https://dotnetcli.blob.core.windows.net/dotnet/checksums/8.0.0-sha.txt
```

6. With both the checksum file and the .NET release file downloaded to the same directory, use the `sha512sum -c {file} --ignore-missing` command to validate the downloaded file.

When validation passes, you see the file printed with the **OK** status:

Bash

```
$ sha512sum -c 8.0.0-sha.txt --ignore-missing  
dotnet-sdk-8.0.100-linux-x64.tar.gz: OK
```

If you see the file marked as **FAILED**, the file you downloaded isn't valid and shouldn't be used.

Bash

```
$ sha512sum -c 8.0.0-sha.txt --ignore-missing  
dotnet-sdk-8.0.100-linux-x64.tar.gz: FAILED  
sha512sum: WARNING: 1 computed checksum did NOT match  
sha512sum: 8.0.0-sha.txt: no file was verified
```

Set environment variables system-wide

If you used the instructions in the [Manual install example](#) section, the variables set only apply to your current terminal session. Add them to your shell profile. There are many different shells available for macOS and each has a different profile. For example:

- **Bash Shell:** `~/.profile`, `/etc/profile`
- **Korn Shell:** `~/.kshrc` or `.profile`
- **Z Shell:** `~/.zshrc` or `.zprofile`

Set the following two environment variables in your shell profile:

- `DOTNET_ROOT`

This variable is set to the folder .NET was installed to, such as `$HOME/.dotnet`:

```
Bash
```

```
export DOTNET_ROOT=$HOME/.dotnet
```

- `PATH`

This variable should include both the `DOTNET_ROOT` folder and the `DOTNET_ROOT/tools` folder:

```
Bash
```

```
export PATH=$PATH:$DOTNET_ROOT:$DOTNET_ROOT/tools
```

Arm-based Macs

The following sections describe things you should consider when installing .NET on an Arm-based Mac.

What's supported

The following table describes which versions of .NET are supported on an Arm-based Mac:

 Expand table

| .NET Version | Architecture | SDK | Runtime | Path conflict |
|--------------|--------------|-----|---------|---------------|
| 8 | Arm64 | Yes | Yes | No |
| 8 | x64 | Yes | Yes | No |
| 7 | Arm64 | Yes | Yes | No |
| 7 | x64 | Yes | Yes | No |
| 6 | Arm64 | Yes | Yes | No |
| 6 | x64 | Yes | Yes | No |

Starting with .NET 6, the x64 and Arm64 versions of the .NET SDK exist independently from each other. If a new version is released, each install needs to be upgraded.

Path differences

On an Arm-based Mac, all Arm64 versions of .NET are installed to the normal `/usr/local/share/dotnet/` folder. However, when you install the x64 version of .NET SDK, it's installed to the `/usr/local/share/dotnet/x64/dotnet/` folder.

Path conflicts

Starting with .NET 6, the x64 .NET SDK installs to its own directory, as described in the previous section. This allows the Arm64 and x64 versions of the .NET SDK to exist on the same machine. However, any x64 SDK prior to .NET 6 isn't supported and installs to the same location as the Arm64 version, the `/usr/local/share/dotnet/` folder. If you want to install an unsupported x64 SDK, you need to first uninstall the Arm64 version. The opposite is also true, you need to uninstall the unsupported x64 SDK to install the Arm64 version.

Path variables

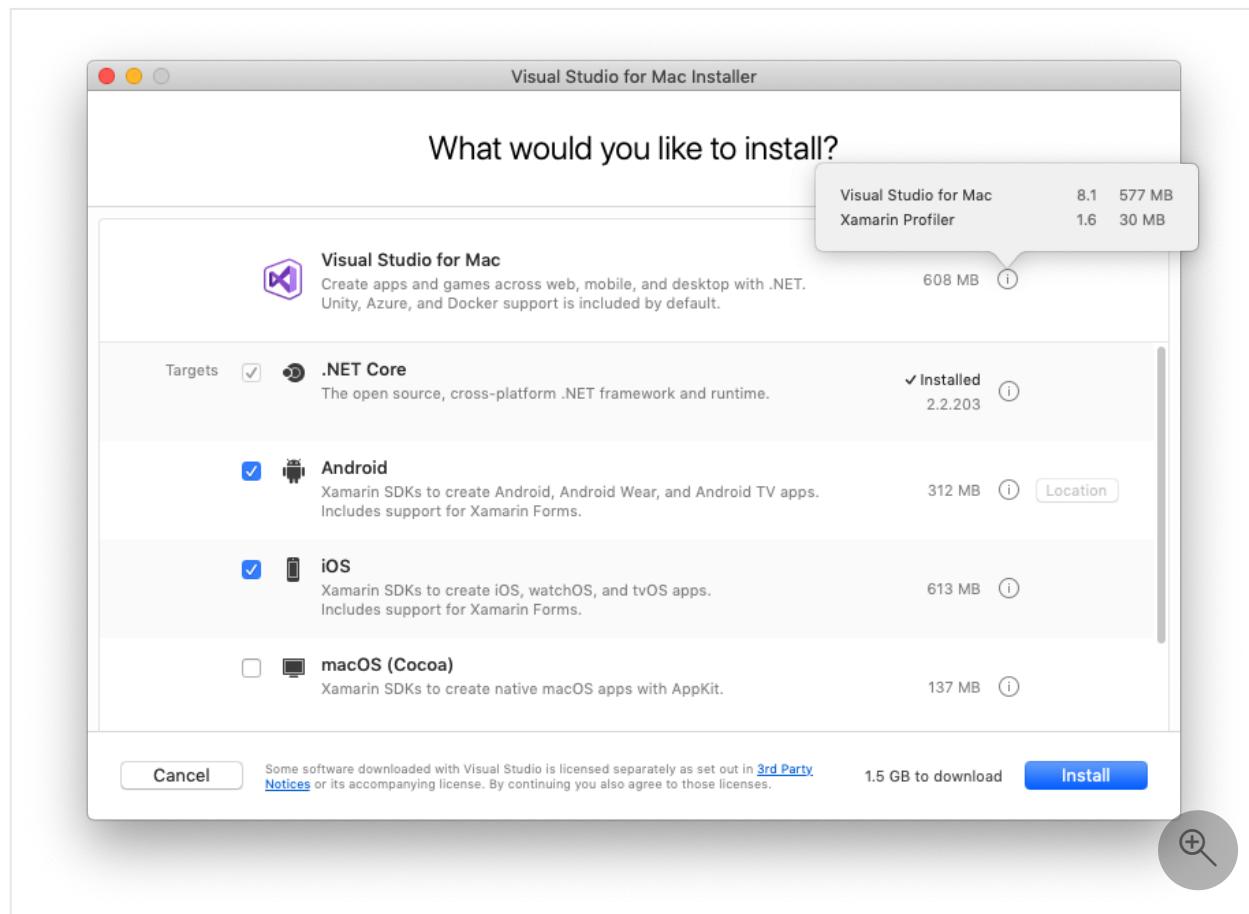
Environment variables that add .NET to system path, such as the `PATH` variable, might need to be changed if you have both the x64 and Arm64 versions of the .NET 6 SDK installed. Additionally, some tools rely on the `DOTNET_ROOT` environment variable, which would also need to be updated to point to the appropriate .NET 6 SDK installation folder.

Install with Visual Studio for Mac

Visual Studio for Mac installs the .NET SDK when the .NET workload is selected. To get started with .NET development on macOS, see [Install Visual Studio 2019 for Mac](#).

[+] Expand table

| .NET SDK version | Visual Studio version |
|------------------|--|
| 8.0 | Visual Studio 2022 for Mac 17.8 or higher. |
| 7.0 | Visual Studio 2022 for Mac 17.4 or higher. |
| 6.0 | Visual Studio 2022 for Mac Preview 3 17.0 or higher. |



ⓘ Important

Microsoft has announced the retirement of Visual Studio for Mac. Visual Studio for Mac will no longer be supported starting August 31, 2024. Alternatives include:

- Visual Studio Code with the [C# Dev Kit](#) and related extensions, such as [.NET MAUI](#) and [Unity](#).
- Visual Studio running on Windows in a VM on Mac.
- Visual Studio running on Windows in a [VM in the Cloud](#).

For more information, see [Visual Studio for Mac retirement announcement](#).

Install alongside Visual Studio Code

Visual Studio Code is a powerful and lightweight source code editor that runs on your desktop. Visual Studio Code is available for Windows, macOS, and Linux.

While Visual Studio Code doesn't come with an automated .NET installer like Visual Studio does, adding .NET support is simple.

1. [Download and install Visual Studio Code](#).
2. [Download and install the .NET SDK](#).
3. [Install the C# extension from the Visual Studio Code marketplace](#).

Install with bash automation

The [dotnet-install scripts](#) are used for automation and non-admin installs of the runtime. You can download the script from the [dotnet-install script reference page](#).

The script defaults to installing the latest [long term support \(LTS\)](#) version, which is .NET 8. You can choose a specific release by specifying the `--channel` switch. Include the `--runtime` switch to install a runtime. Otherwise, the script installs the SDK.

The following command installs the ASP.NET Core runtime for maximum compatibility. The ASP.NET Core runtime also includes the standard .NET runtime.

Bash

```
./dotnet-install.sh --channel 8.0 --runtime aspnetcore
```

Docker

Containers provide a lightweight way to isolate your application from the rest of the host system. Containers on the same machine share just the kernel and use resources given to your application.

.NET can run in a Docker container. Official .NET Docker images are published to the Microsoft Container Registry (MCR) and are discoverable at the [Microsoft .NET Docker Hub repository](#). Each repository contains images for different combinations of the .NET (SDK or Runtime) and OS that you can use.

Microsoft provides images that are tailored for specific scenarios. For example, the [ASP.NET Core repository](#) provides images that are built for running ASP.NET Core apps in production.

For more information about using .NET in a Docker container, see [Introduction to .NET and Docker](#) and [Samples](#).

Next steps

- [How to check if .NET is already installed.](#)
- [Working with macOS Catalina notarization.](#)
- [Tutorial: Get started on macOS.](#)
- [Tutorial: Create a new app with Visual Studio Code.](#)
- [Tutorial: Containerize a .NET app.](#)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



.NET feedback

.NET is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

Install .NET on Linux

Article • 12/15/2023

This article details how to install .NET on various Linux distributions either manually, via a package manager, or via a [container](#).

Manual installation

You can install .NET manually in the following ways:

- [Manual install](#)
- [Scripted install](#)

You may need to install [.NET dependencies](#) if you install .NET manually.

Packages

.NET is available in [official package archives](#) for various Linux distributions and [packages.microsoft.com](#).

- [Alpine](#)
- [CentOS](#)
- [Debian](#)
- [Fedora](#)
- [openSUSE](#)
- [SLES](#)
- [Ubuntu](#)

.NET is [supported by Microsoft](#) when downloaded from a Microsoft source. Best effort support is offered from Microsoft when downloaded from elsewhere. You can open issues at [dotnet/core](#) if you run into problems.

Next steps

- [How to check if .NET is already installed.](#)
- [Tutorial: Create a new app with Visual Studio Code.](#)
- [Tutorial: Containerize a .NET app.](#)

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

.NET is an open source project.
Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

Install the .NET SDK or the .NET Runtime on Ubuntu

Article • 11/14/2023

This article describes how to install .NET on Ubuntu. The Microsoft package repository contains every version of .NET that is currently, or was previously, supported on Ubuntu. Starting with Ubuntu 22.04, some versions of .NET are available in the Ubuntu package feed. For more information about available versions, see the [Supported distributions](#) section.

⚠ Warning

It's recommended that you choose a single repository to source .NET packages. Don't mix .NET packages from multiple package repositories, as this leads to problems when apps try to resolve a specific version of .NET.

[+] Expand table

| Method | Pros | Cons |
|--|--|--|
| Package manager (Microsoft feed) | <ul style="list-style-type: none">Supported versions always available.Patches are available right way.Dependencies are included.Easy removal. | <ul style="list-style-type: none">Requires registering the Microsoft package repository.Preview releases aren't available.Only supports x64 Ubuntu. |
| Package manager (Ubuntu feed) | <ul style="list-style-type: none">Usually the latest version is available.Patches are available right way.Dependencies are included.Easy removal. | <ul style="list-style-type: none">.NET versions available vary by Ubuntu version.Preview releases aren't available.Only supports x64 Ubuntu. (Except for Ubuntu 23.04+, which also supports Arm64) |
| Script \ Manual extraction | <ul style="list-style-type: none">Control where .NET is installed.Preview releases are available. | <ul style="list-style-type: none">Manually install updates.Manually install dependencies.Manual removal. |

Decide how to install .NET

When your version of Ubuntu supports .NET through the built-in Ubuntu feed, support for those builds of .NET is provided by Canonical and the builds might be optimized for different workloads. Microsoft provides support for packages in the Microsoft package repository feed.

Use the following sections to determine how you should install .NET:

- I'm using Ubuntu 22.04 or later, and I only need .NET
- I'm using a version of Ubuntu prior to 22.04
- I'm using other Microsoft packages, such as powershell, mdatp, or mssql
- I want to create a .NET app
- I want to run a .NET app in a container, cloud, or continuous-integration scenario
- My Ubuntu distribution doesn't include the .NET version I want, or I need an out-of-support .NET version
- I want to install a preview version
- I don't want to use APT
- I'm using an Arm-based CPU

I'm using Ubuntu 22.04 or later, and I only need .NET

Install .NET through the Ubuntu feed. For more information, see the following pages:

- [Install .NET on Ubuntu 22.04.](#)
- [Install .NET on Ubuntu 22.10.](#)
- [Install .NET on Ubuntu 23.04.](#)
- [Install .NET on Ubuntu 23.10.](#)

Important

.NET SDK versions offered by Canonical are always in the [.1xx feature band](#). If you want to use a newer feature band release, use the [Microsoft feed to install the SDK](#). Make sure you review the information in the [.NET package mix ups on Linux](#) article to understand the implications of switching between repository feeds.

If you're going to install the Microsoft repository to use other Microsoft packages, such as `powershell`, `mdatp`, or `mssql`, you need to deprioritize the .NET packages provided by the Microsoft repository. For instructions on how to deprioritize the packages, see [My Linux distribution provides .NET packages, and I want to use them](#).

I'm using a version of Ubuntu prior to 22.04

Use the instructions on the version-specific Ubuntu page.

- [20.04 \(LTS\)](#)
- [18.04 \(LTS\)](#)
- [16.04 \(LTS\)](#)

Review the [Supported distributions](#) section for more information about what versions of .NET are supported for your version of Ubuntu. If you're installing a version that isn't supported, see [Register the Microsoft package repository](#).

I'm using other Microsoft packages, such as `powershell`, `mdatp`, or `mssql`

If your Ubuntu version supports .NET through the built-in Ubuntu feed, you must decide which feed should install .NET. The [Supported distributions](#) section provides a table that lists which versions of .NET are available the package feeds.

If you want to source the .NET packages from the Ubuntu feed, you need to deprioritize the .NET packages provided by the Microsoft repository. For instructions on how to deprioritize the packages, see [My Linux distribution provides .NET packages, and I want to use them](#).

I want to create a .NET app

Use the same package sources for the SDK as you use for the runtime. For example, if you're using Ubuntu 22.04 and .NET 6, but not .NET 7, it's recommended that you install .NET through the built-in Ubuntu feed. If, however, you move to .NET 7, which isn't provided by Canonical for Ubuntu 22.04, you should uninstall .NET and reinstall it with the [Microsoft package repository](#). For more information, see [Register and install with the Microsoft package repository](#). Also, review the other suggestions in the [Decide how to install .NET](#) section.

I want to run a .NET app in a container, cloud, or continuous-integration scenario

If your Ubuntu version provides the .NET version you require, install it from the built-in feed. Otherwise, [register the Microsoft package repository](#) and install .NET from that repository. Review the information in the [Supported distributions](#) section.

If the version of .NET you want isn't available, try using the [dotnet-install](#) script.

My Ubuntu distribution doesn't include the .NET version I want, or I need an out-of-support .NET version

We recommend you use APT and the Microsoft package repository. For more information, see the [Register and install with the Microsoft package repository](#) section.

I want to install a preview version

Use one of the following ways to install .NET:

- [Install .NET with install-dotnet script.](#)
- [Manually install .NET](#)

I don't want to use APT

If you want an automated installation, use the [Linux installation script](#).

If you want full control over the .NET installation experience, download a tarball and manually install .NET. For more information, see [Manual install](#).

I'm using an Arm-based CPU

Use one of the following ways to install .NET:

- [Install .NET with install-dotnet script.](#)
- [Manually install .NET](#)

Supported distributions

The following table is a list of currently supported .NET releases and the versions of Ubuntu they're supported on. Each link goes to the specific Ubuntu version page with specific instructions on how to install .NET for that version of Ubuntu.

[+] Expand table

| Ubuntu | Supported .NET versions | Available in Ubuntu feed | Available in Microsoft feed |
|-----------------------|-------------------------|--------------------------|-----------------------------|
| 23.10 | 8.0, 7.0, 6.0 | 8.0, 7.0, 6.0 | 8.0, 7.0, 6.0 |

| Ubuntu | Supported .NET versions | Available in Ubuntu feed | Available in Microsoft feed |
|----------------|--------------------------------|---------------------------------|------------------------------------|
| 23.04 | 8.0, 7.0, 6.0 | 7.0, 6.0 | 8.0, 7.0, 6.0 |
| 22.10 | 7.0, 6.0 | 7.0, 6.0 | 7.0, 6.0, 3.1 |
| 22.04 (LTS) | 8.0, 7.0, 6.0 | 6.0 | 8.0, 7.0, 6.0, 3.1 |
| 20.04 (LTS) | 8.0, 7.0, 6.0 | None | 8.0, 7.0, 6.0, 5.0, 3.1, 2.1 |
| 18.04 (LTS) | 7.0, 6.0 | None | 7.0, 6.0, 5.0, 3.1, 2.2, 2.1 |
| 16.04 (LTS) | 6.0 | None | 6.0, 5.0, 3.1, 3.0, 2.2, 2.1, 2.0 |

When an [Ubuntu version ↗](#) falls out of support, .NET is no longer supported with that version.

The following versions of .NET are  no longer supported:

- .NET 5
- .NET Core 3.1
- .NET Core 3.0
- .NET Core 2.2
- .NET Core 2.1
- .NET Core 2.0

Register the Microsoft package repository

The Microsoft package repository contains all versions of .NET that were previously, or are currently, [supported with your version of Ubuntu](#). If your version of Ubuntu provides .NET packages, you'll need to deprioritize the Ubuntu packages and use the Microsoft repository. For instructions on how to deprioritize the packages, see [I need a version of .NET that isn't provided by my Linux distribution](#).

Important

Package manager installs are only supported on the **x64** architecture. Other architectures, such as **Arm**, must install .NET by some other means, such as with the [installer script](#) or by [manual installation](#).

Preview releases are **not** available in the Microsoft package repository. For more information, see [Install preview versions](#).

⊗ Caution

We recommend that you only use one repository to manage all of your .NET installs. If you've previously installed .NET with the Ubuntu repository, you must clean the system of .NET packages and configure the APT to ignore the Ubuntu feed. For more information about how to do this, see [I need a version of .NET that isn't provided by my Linux distribution](#).

Installing with APT can be done with a few commands. Before you install .NET, run the following commands to add the Microsoft package signing key to your list of trusted keys and add the package repository.

Open a terminal and run the following commands:

Bash

```
# Get Ubuntu version
declare repo_version=$(if command -v lsb_release &> /dev/null; then
    lsb_release -r -s; else grep -oP '(?=<^VERSION_ID=).+' /etc/os-release | tr
    -d '''; fi)

# Download Microsoft signing key and repository
wget https://packages.microsoft.com/config/ubuntu/$repo_version/packages-
microsoft-prod.deb -O packages-microsoft-prod.deb

# Install Microsoft signing key and repository
sudo dpkg -i packages-microsoft-prod.deb

# Clean up
rm packages-microsoft-prod.deb

# Update packages
sudo apt update
```

💡 Tip

The previous script was written for Ubuntu and it might not work if you're using a derived distribution, such as Linux Mint. It's likely that the `$repo_version` variable won't be assigned the correct value, making the URI for the `wget` command invalid. This variable maps to the specific Ubuntu version you want to get packages for, such as 22.10 or 23.04.

You can use a web browser and navigate to <https://packages.microsoft.com/config/ubuntu/> to see which versions of Ubuntu are available to use as the `$repo_version` value.

Install .NET

After you've [registered the Microsoft package repository](#), or if your version of Ubuntu's default feed supports the .NET package, you can install .NET through the package manager with the `sudo apt install <package-name>` command. Replace `<package-name>` with the name of the .NET package you want to install. For example, to install .NET SDK 8.0, use the command `sudo apt install dotnet-sdk-8.0`. The following table lists the currently supported .NET packages:

[+] Expand table

| Product | Type | Package |
|---------|--------------|------------------------|
| 8.0 | ASP.NET Core | aspnetcore-runtime-8.0 |
| 8.0 | .NET | dotnet-runtime-8.0 |
| 8.0 | .NET | dotnet-sdk-8.0 |
| 7.0 | ASP.NET Core | aspnetcore-runtime-7.0 |
| 7.0 | .NET | dotnet-runtime-7.0 |
| 7.0 | .NET | dotnet-sdk-7.0 |
| 6.0 | ASP.NET Core | aspnetcore-runtime-6.0 |
| 6.0 | .NET | dotnet-runtime-6.0 |
| 6.0 | .NET | dotnet-sdk-6.0 |

If you want to install an unsupported version of .NET, check the [Supported distributions](#) section to see if that version of .NET is available. Then, substitute the [version](#) of .NET you want to install. For example, to install ASP.NET Core 2.1, use the package name `aspnetcore-runtime-2.1`.

💡 Tip

If you're not creating .NET apps, install the ASP.NET Core runtime as it includes the .NET runtime and also supports ASP.NET Core apps.

Some environment variables affect how .NET is run after it's installed. For more information, see [.NET SDK and CLI environment variables](#).

Uninstall .NET

If you installed .NET through a package manager, uninstall in the same way with the `apt-get remove` command:

Bash

```
sudo apt-get remove dotnet-sdk-6.0
```

For more information, see [Uninstall .NET](#).

Install preview versions

Preview and release candidate versions of .NET aren't available in package repositories. You can install previews and release candidates of .NET in one of the following ways:

- Scripted install with [*install-dotnet.sh*](#)
- Manual binary extraction

Remove preview versions

When using a package manager to manage your installation of .NET, you may run into a conflict if you've previously installed a preview release. The package manager may interpret the non-preview release as an earlier version of .NET. To install the non-preview release, first uninstall the preview versions. For more information about uninstalling .NET, see [How to remove the .NET Runtime and SDK](#).

Use APT to update .NET

If you installed .NET through a package manager, you can upgrade the package with the `apt upgrade` command. For example, the following commands upgrade the `dotnet-sdk-7.0` package with the latest version:

Bash

```
sudo apt update  
sudo apt upgrade dotnet-sdk-7.0
```

💡 Tip

If you've upgraded your Linux distribution since installing .NET, you may need to reconfigure the Microsoft package repository. Run the installation instructions for your current distribution version to upgrade to the appropriate package repository for .NET updates.

Troubleshooting

Starting with Ubuntu 22.04, you might run into a situation where it seems only a piece of .NET is available. For example, you've installed the runtime and the SDK, but when you run `dotnet --info` only the runtime is listed. This situation can be related to using two different package sources. The built-in Ubuntu 22.04 and Ubuntu 22.10 package feeds include some versions of .NET, but not all, and you might have also installed .NET from the Microsoft feeds. For more information about how to fix this problem, see [Troubleshoot .NET errors related to missing files on Linux](#).

APT problems

This section provides information on common errors you might get while using APT to install .NET.

Unable to find package

ⓘ Important

Package manager installs are only supported on the **x64** architecture. Other architectures, such as **Arm**, must install .NET by some other means such as with Snap, an installer script, or through a manual binary installation.

For more information on installing .NET **without a package manager**, see one of the following articles:

- [Alternatively install .NET with install-dotnet script](#).
- [Manually install .NET](#)

Unable to locate \ Some packages could not be installed

ⓘ Note

This information only applies when .NET is installed from the Microsoft package feed.

If you receive an error message similar to **Unable to locate package {dotnet-package}** or **Some packages could not be installed**, run the following commands.

There are two placeholders in the following set of commands.

- `{dotnet-package}`

This represents the .NET package you're installing, such as `aspnetcore-runtime-8.0`. This is used in the following `sudo apt-get install` command.

- `{os-version}`

This represents the distribution version you're on. This is used in the `wget` command below. The distribution version is the numerical value, such as `20.04` on Ubuntu or `10` on Debian.

First, try purging the package list:

Bash

```
sudo dpkg --purge packages-microsoft-prod && sudo dpkg -i packages-microsoft-prod.deb  
sudo apt-get update
```

Then, try to install .NET again. If that doesn't work, you can run a manual install with the following commands:

Bash

```
sudo apt-get install -y gpg  
wget -O - https://packages.microsoft.com/keys/microsoft.asc | gpg --dearmor  
-o microsoft.asc.gpg  
sudo mv microsoft.asc.gpg /etc/apt/trusted.gpg.d/  
wget https://packages.microsoft.com/config/ubuntu/{os-version}/prod.list  
sudo mv prod.list /etc/apt/sources.list.d/microsoft-prod.list  
sudo chown root:root /etc/apt/trusted.gpg.d/microsoft.asc.gpg  
sudo chown root:root /etc/apt/sources.list.d/microsoft-prod.list  
sudo apt-get update && \  
sudo apt-get install -y {dotnet-package}
```

Failed to fetch

While installing the .NET package, you may see an error similar to `Failed to fetch ... File has unexpected size ... Mirror sync in progress?`. This error could mean that the package feed for .NET is being upgraded with newer package versions, and that you should try again later. During an upgrade, the package feed shouldn't be unavailable for more than 30 minutes. If you continually receive this error for more than 30 minutes, please file an issue at <https://github.com/dotnet/core/issues>.

Dependencies

When you install with a package manager, these libraries are installed for you. But, if you manually install .NET or you publish a self-contained app, you must install these dependencies to run your app:

- libc6
- libgcc1
- libgcc-s1 (for 22.x)
- libgssapi-krb5-2
- libicu55 (for 16.x)
- libicu60 (for 18.x)
- libicu66 (for 20.x)
- libicu70 (for 22.04)
- libicu71 (for 22.10)
- libicu72 (for 23.04)
- liblttng-ust1 (for 22.x)
- libssl1.0.0 (for 16.x)
- libssl1.1 (for 18.x, 20.x)
- libssl3 (for 22.x)
- libstdc++6
- libunwind8 (for 22.x)
- zlib1g

Dependencies can be installed with the `apt install` command. The following snippet demonstrates installing the `zlib1g` library:

Bash

```
sudo apt install zlib1g
```

If the .NET app uses the `System.Drawing.Common` assembly, libgdiplus will also need to be installed. Because [System.Drawing.Common is no longer supported on Linux](#), this only works on .NET 6 and requires setting the `System.Drawing.EnableUnixSupport` runtime configuration switch.

You can install a recent version of *libgdiplus* by [adding the Mono repository to your system ↗](#).

Next steps

- [How to enable Tab completion for the .NET CLI](#)
- [Tutorial: Create a console application with .NET SDK using Visual Studio Code](#)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



.NET feedback

.NET is an open source project. Select a link to provide feedback:

-  [Open a documentation issue](#)
-  [Provide product feedback](#)

Install .NET SDK or .NET Runtime on Ubuntu 23.10

Article • 12/30/2023

This article discusses how to install .NET on Ubuntu 23.10; .NET 8, .NET 7, and .NET 6, are supported.

Install the SDK (which includes the runtime) if you want to develop .NET apps. Or, if you only need to run apps, install the Runtime. If you're installing the Runtime, we suggest you install the **ASP.NET Core Runtime** as it includes both .NET and ASP.NET Core runtimes.

Use the `dotnet --list-sdks` and `dotnet --list-runtimes` commands to see which versions are installed. For more information, see [How to check that .NET is already installed](#).

Important

Using a package manager to install .NET from the **Microsoft pacakge feed** only supports the **x64** architecture. Other architectures, such as **Arm**, aren't supported by the **Microsoft package feed**. Use the Ubuntu feed or manually install .NET. Be cautious of package mix up problems. For more information, see [.NET package mix ups on Linux](#).

For more information on installing .NET **without a package manager**, see one of the following articles:

- [Use the install-dotnet script to install .NET](#).
- [Manually install .NET](#).

.NET is available in the Ubuntu package manager feeds, as well as the Microsoft package repository. However, you should only use one or the other to install .NET. If you want to use the Microsoft package repository, see [How to register the Microsoft package repository](#).

Supported versions

The following versions of .NET are supported or available for Ubuntu 23.10:

| Supported .NET versions | Available in Ubuntu feed | Available in Microsoft feed |
|-------------------------|--------------------------|-----------------------------|
| 8.0, 7.0, 6.0 | 8.0, 7.0, 6.0 | 8.0, 7.0, 6.0 |

ⓘ Important

.NET SDK versions offered by Canonical are always in the **.1xx feature band**. If you want to use a newer feature band release, use the [Microsoft feed to install the SDK](#). Make sure you review the information in the [.NET package mix ups on Linux](#) article to understand the implications of switching between repository feeds.

When an [Ubuntu version ↗](#) falls out of support, .NET is no longer supported with that version.

The following versions of .NET are **✗** no longer supported:

- .NET 5
- .NET Core 3.1
- .NET Core 3.0
- .NET Core 2.2
- .NET Core 2.1
- .NET Core 2.0

Install .NET

Install the SDK

The .NET SDK allows you to develop apps with .NET. If you install the .NET SDK, you don't need to install the corresponding runtime. To install the .NET SDK, run the following commands:

Bash

```
sudo apt-get update && \
  sudo apt-get install -y dotnet-sdk-8.0
```

ⓘ Important

If you receive an error message similar to **Unable to locate package dotnet-sdk-8.0**, see the [troubleshooting](#) section.

Install the runtime

The ASP.NET Core Runtime allows you to run apps that were made with .NET that didn't provide the runtime. The following commands install the ASP.NET Core Runtime, which is the most compatible runtime for .NET. In your terminal, run the following commands:

Bash

```
sudo apt-get update && \
  sudo apt-get install -y aspnetcore-runtime-8.0
```

ⓘ Important

If you receive an error message similar to **Unable to locate package aspnetcore-runtime-8.0**, see the [troubleshooting](#) section.

As an alternative to the ASP.NET Core Runtime, you can install the .NET Runtime, which doesn't include ASP.NET Core support: replace `aspnetcore-runtime-8.0` in the previous command with `dotnet-runtime-8.0`:

Bash

```
sudo apt-get install -y dotnet-runtime-8.0
```

How to install other versions

.NET package names are standardized across all Linux distributions. The following table lists the packages:

All versions of .NET are available for download at <https://dotnet.microsoft.com/download/dotnet>, but require [manual installation](#). You can try and use the package manager to install a different version of .NET. However, the requested version may not be available.

The packages added to package manager feeds are named in a hackable format, for example: `{product}-{type}-{version}`.

- **product**

The type of .NET product to install. Valid options are:

- dotnet
- aspnetcore

- **type**

Chooses the SDK or the runtime. Valid options are:

- sdk (only available for the **dotnet** product)
- runtime

- **version**

The version of the SDK or runtime to install. This article will always give the instructions for the latest supported version. Valid options are any released version, such as:

- 8.0
- 6.0
- 3.1
- 2.1

It's possible the SDK/runtime you're trying to download is not available for your Linux distribution. For a list of supported distributions, see [Install .NET on Linux](#).

Examples

- Install the ASP.NET Core 8.0 runtime: `aspnetcore-runtime-8.0`
- Install the .NET Core 2.1 runtime: `dotnet-runtime-2.1`
- Install the .NET 5 SDK: `dotnet-sdk-5.0`
- Install the .NET Core 3.1 SDK: `dotnet-sdk-3.1`

Package missing

If the package-version combination doesn't work, it's not available. For example, there isn't an ASP.NET Core SDK, the SDK components are included with the .NET SDK. The value `aspnetcore-sdk-8.0` is incorrect and should be `dotnet-sdk-8.0`. For a list of Linux distributions supported by .NET, see [.NET dependencies and requirements](#).

Troubleshooting

If you run into issues installing or even running .NET, see [Troubleshooting](#).

Dependencies

When you install with a package manager, these libraries are installed for you. But, if you manually install .NET or you publish a self-contained app, you'll need to make sure these libraries are installed:

- libc6
- libgcc1
- libgcc-s1
- libgssapi-krb5-2
- libicu72
- liblttng-ust1
- libssl3
- libstdc++6
- libunwind8
- zlib1g

Dependencies can be installed with the `apt install` command. The following snippet demonstrates installing the `zlib1g` library:

```
Bash
```

```
sudo apt install zlib1g
```

If the .NET app uses the `System.Drawing.Common` assembly, `libgdiplus` will also need to be installed. Because [System.Drawing.Common is no longer supported on Linux](#), this only works on .NET 6 and requires setting the `System.Drawing.EnableUnixSupport` runtime configuration switch.

You can install a recent version of `libgdiplus` by [adding the Mono repository to your system ↗](#).

Next steps

- [How to enable TAB completion for the .NET CLI](#)
- [Tutorial: Create a console application with .NET SDK using Visual Studio Code](#)

 Collaborate with us on
GitHub



.NET feedback

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET is an open source project.
Select a link to provide feedback:

-  [Open a documentation issue](#)
-  [Provide product feedback](#)

Install .NET SDK or .NET Runtime on Ubuntu 23.04

Article • 12/30/2023

This article discusses how to install .NET on Ubuntu 23.04; .NET 8, .NET 7, and .NET 6, are supported.

Install the SDK (which includes the runtime) if you want to develop .NET apps. Or, if you only need to run apps, install the Runtime. If you're installing the Runtime, we suggest you install the **ASP.NET Core Runtime** as it includes both .NET and ASP.NET Core runtimes.

Use the `dotnet --list-sdks` and `dotnet --list-runtimes` commands to see which versions are installed. For more information, see [How to check that .NET is already installed](#).

Important

Using a package manager to install .NET from the **Microsoft pacakge feed** only supports the **x64** architecture. Other architectures, such as **Arm**, aren't supported by the **Microsoft package feed**. Use the Ubuntu feed or manually install .NET. Be cautious of package mix up problems. For more information, see [.NET package mix ups on Linux](#).

For more information on installing .NET **without a package manager**, see one of the following articles:

- [Use the install-dotnet script to install .NET](#).
- [Manually install .NET](#).

.NET is available in the Ubuntu package manager feeds, as well as the Microsoft package repository. However, you should only use one or the other to install .NET. If you want to use the Microsoft package repository, see [How to register the Microsoft package repository](#).

Supported versions

The following versions of .NET are supported or available for Ubuntu 23.04:

| Supported .NET versions | Available in Ubuntu feed | Available in Microsoft feed |
|-------------------------|--------------------------|-----------------------------|
| 8.0, 7.0, 6.0 | 7.0, 6.0 | 8.0, 7.0, 6.0 |

 ⓘ Important

.NET SDK versions offered by Canonical are always in the **.1xx feature band**. If you want to use a newer feature band release, use the [Microsoft feed to install the SDK](#). Make sure you review the information in the [.NET package mix ups on Linux](#) article to understand the implications of switching between repository feeds.

When an [Ubuntu version ↗](#) falls out of support, .NET is no longer supported with that version.

The following versions of .NET are **✗** no longer supported:

- .NET 5
- .NET Core 3.1
- .NET Core 3.0
- .NET Core 2.2
- .NET Core 2.1
- .NET Core 2.0

Install .NET

Install the SDK

The .NET SDK allows you to develop apps with .NET. If you install the .NET SDK, you don't need to install the corresponding runtime. To install the .NET SDK, run the following commands:

Bash

```
sudo apt-get update && \
  sudo apt-get install -y dotnet-sdk-8.0
```

 ⓘ Important

If you receive an error message similar to **Unable to locate package dotnet-sdk-8.0**, see the [troubleshooting](#) section.

Install the runtime

The ASP.NET Core Runtime allows you to run apps that were made with .NET that didn't provide the runtime. The following commands install the ASP.NET Core Runtime, which is the most compatible runtime for .NET. In your terminal, run the following commands:

Bash

```
sudo apt-get update && \
  sudo apt-get install -y aspnetcore-runtime-8.0
```

ⓘ Important

If you receive an error message similar to **Unable to locate package aspnetcore-runtime-8.0**, see the [troubleshooting](#) section.

As an alternative to the ASP.NET Core Runtime, you can install the .NET Runtime, which doesn't include ASP.NET Core support: replace `aspnetcore-runtime-8.0` in the previous command with `dotnet-runtime-8.0`:

Bash

```
sudo apt-get install -y dotnet-runtime-8.0
```

How to install other versions

.NET package names are standardized across all Linux distributions. The following table lists the packages:

All versions of .NET are available for download at <https://dotnet.microsoft.com/download/dotnet>, but require [manual installation](#). You can try and use the package manager to install a different version of .NET. However, the requested version may not be available.

The packages added to package manager feeds are named in a hackable format, for example: `{product}-{type}-{version}`.

- **product**

The type of .NET product to install. Valid options are:

- dotnet
- aspnetcore

- **type**

Chooses the SDK or the runtime. Valid options are:

- sdk (only available for the **dotnet** product)
- runtime

- **version**

The version of the SDK or runtime to install. This article will always give the instructions for the latest supported version. Valid options are any released version, such as:

- 8.0
- 6.0
- 3.1
- 2.1

It's possible the SDK/runtime you're trying to download is not available for your Linux distribution. For a list of supported distributions, see [Install .NET on Linux](#).

Examples

- Install the ASP.NET Core 8.0 runtime: `aspnetcore-runtime-8.0`
- Install the .NET Core 2.1 runtime: `dotnet-runtime-2.1`
- Install the .NET 5 SDK: `dotnet-sdk-5.0`
- Install the .NET Core 3.1 SDK: `dotnet-sdk-3.1`

Package missing

If the package-version combination doesn't work, it's not available. For example, there isn't an ASP.NET Core SDK, the SDK components are included with the .NET SDK. The value `aspnetcore-sdk-8.0` is incorrect and should be `dotnet-sdk-8.0`. For a list of Linux distributions supported by .NET, see [.NET dependencies and requirements](#).

Troubleshooting

If you run into issues installing or even running .NET, see [Troubleshooting](#).

Dependencies

When you install with a package manager, these libraries are installed for you. But, if you manually install .NET or you publish a self-contained app, you'll need to make sure these libraries are installed:

- libc6
- libgcc1
- libgcc-s1
- libgssapi-krb5-2
- libicu72
- liblttng-ust1
- libssl3
- libstdc++6
- libunwind8
- zlib1g

Dependencies can be installed with the `apt install` command. The following snippet demonstrates installing the `zlib1g` library:

```
Bash
```

```
sudo apt install zlib1g
```

If the .NET app uses the `System.Drawing.Common` assembly, `libgdiplus` will also need to be installed. Because [System.Drawing.Common is no longer supported on Linux](#), this only works on .NET 6 and requires setting the `System.Drawing.EnableUnixSupport` runtime configuration switch.

You can install a recent version of `libgdiplus` by [adding the Mono repository to your system ↗](#).

Next steps

- [How to enable TAB completion for the .NET CLI](#)
- [Tutorial: Create a console application with .NET SDK using Visual Studio Code](#)



Collaborate with us on
GitHub



.NET feedback

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET is an open source project.
Select a link to provide feedback:

-  [Open a documentation issue](#)
-  [Provide product feedback](#)

Install .NET SDK or .NET Runtime on Ubuntu 22.10

Article • 05/25/2023

This article discusses how to install .NET on Ubuntu 22.10; .NET 6 and .NET 7 are both supported.

Install the SDK (which includes the runtime) if you want to develop .NET apps. Or, if you only need to run apps, install the Runtime. If you're installing the Runtime, we suggest you install the **ASP.NET Core Runtime** as it includes both .NET and ASP.NET Core runtimes.

Use the `dotnet --list-sdks` and `dotnet --list-runtimes` commands to see which versions are installed. For more information, see [How to check that .NET is already installed](#).

Important

Package manager installs are only supported on the **x64** architecture. Other architectures, such as **Arm**, must install .NET by some other means such as with Snap, an installer script, or through a manual binary installation.

For more information on installing .NET **without a package manager**, see one of the following articles:

- [Alternatively install .NET with install-dotnet script.](#)
- [Manually install .NET](#)

.NET is available in the Ubuntu package manager feeds, as well as the Microsoft package repository. However, you should only use one or the other to install .NET. If you want to use the Microsoft package repository, see [How to register the Microsoft package repository](#).

Warning

Don't use both repositories to manage .NET. If you've previously installed .NET from the Ubuntu feed or the Microsoft feed, you'll run into issues using the other feed. .NET is installed to different locations and is resolved differently for both package feeds. It's recommended that you uninstall previously installed versions of .NET and

then install with the Microsoft package repository. For more information, see [How to register the Microsoft package repository](#).

Supported versions

The following versions of .NET are supported or available for Ubuntu 22.10:

| Supported .NET versions | Available in Ubuntu feed | Available in Microsoft feed |
|-------------------------|--------------------------|-----------------------------|
| 7.0, 6.0 | 7.0, 6.0 | 7.0, 6.0, 3.1 |

ⓘ Important

.NET SDK versions offered by Canonical are always in the [.1xx feature band](#). If you want to use a newer feature band release, use the [Microsoft feed to install the SDK](#). Make sure you review the information in the [.NET package mix ups on Linux](#) article to understand the implications of switching between repository feeds.

When an [Ubuntu version](#) falls out of support, .NET is no longer supported with that version.

The following versions of .NET are  no longer supported:

- .NET 5
- .NET Core 3.1
- .NET Core 3.0
- .NET Core 2.2
- .NET Core 2.1
- .NET Core 2.0

Install .NET

Install the SDK

The .NET SDK allows you to develop apps with .NET. If you install the .NET SDK, you don't need to install the corresponding runtime. To install the .NET SDK, run the following commands:

Bash

```
sudo apt-get update && \
    sudo apt-get install -y dotnet-sdk-7.0
```

ⓘ Important

If you receive an error message similar to **Unable to locate package dotnet-sdk-7.0**, see the [troubleshooting](#) section.

Install the runtime

The ASP.NET Core Runtime allows you to run apps that were made with .NET that didn't provide the runtime. The following commands install the ASP.NET Core Runtime, which is the most compatible runtime for .NET. In your terminal, run the following commands:

Bash

```
sudo apt-get update && \
    sudo apt-get install -y aspnetcore-runtime-7.0
```

ⓘ Important

If you receive an error message similar to **Unable to locate package aspnetcore-runtime-7.0**, see the [troubleshooting](#) section.

As an alternative to the ASP.NET Core Runtime, you can install the .NET Runtime, which doesn't include ASP.NET Core support: replace `aspnetcore-runtime-7.0` in the previous command with `dotnet-runtime-7.0`:

Bash

```
sudo apt-get install -y dotnet-runtime-7.0
```

How to install other versions

.NET package names are standardized across all Linux distributions. The following table lists the packages:

All versions of .NET are available for download at

<https://dotnet.microsoft.com/download/dotnet> ↗, but require [manual installation](#). You

can try and use the package manager to install a different version of .NET. However, the requested version may not be available.

The packages added to package manager feeds are named in a hackable format, for example: `{product}-{type}-{version}`.

- **product**

The type of .NET product to install. Valid options are:

- dotnet
- aspnetcore

- **type**

Chooses the SDK or the runtime. Valid options are:

- sdk (only available for the **dotnet** product)
- runtime

- **version**

The version of the SDK or runtime to install. This article will always give the instructions for the latest supported version. Valid options are any released version, such as:

- 8.0
- 6.0
- 3.1
- 2.1

It's possible the SDK/runtime you're trying to download is not available for your Linux distribution. For a list of supported distributions, see [Install .NET on Linux](#).

Examples

- Install the ASP.NET Core 8.0 runtime: `aspnetcore-runtime-8.0`
- Install the .NET Core 2.1 runtime: `dotnet-runtime-2.1`
- Install the .NET 5 SDK: `dotnet-sdk-5.0`
- Install the .NET Core 3.1 SDK: `dotnet-sdk-3.1`

Package missing

If the package-version combination doesn't work, it's not available. For example, there isn't an ASP.NET Core SDK, the SDK components are included with the .NET SDK. The value `aspnetcore-sdk-8.0` is incorrect and should be `dotnet-sdk-8.0`. For a list of Linux distributions supported by .NET, see [.NET dependencies and requirements](#).

Troubleshooting

If you run into issues installing or even running .NET, see [Troubleshooting](#).

Dependencies

When you install with a package manager, these libraries are installed for you. But, if you manually install .NET or you publish a self-contained app, you'll need to make sure these libraries are installed:

- libc6
- libgcc1
- libgcc-s1
- libgssapi-krb5-2
- libicu71
- liblttng-ust1
- libssl3
- libstdc++6
- libunwind8
- zlib1g

Dependencies can be installed with the `apt install` command. The following snippet demonstrates installing the `zlib1g` library:

Bash

```
sudo apt install zlib1g
```

If the .NET app uses the `System.Drawing.Common` assembly, `libgdiplus` will also need to be installed. Because [`System.Drawing.Common` is no longer supported on Linux](#), this only works on .NET 6 and requires setting the `System.Drawing.EnableUnixSupport` runtime configuration switch.

You can install a recent version of `libgdiplus` by [adding the Mono repository to your system ↗](#).

Next steps

- [How to enable TAB completion for the .NET CLI](#)
- [Tutorial: Create a console application with .NET SDK using Visual Studio Code](#)

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

Install .NET SDK or .NET Runtime on Ubuntu 22.04

Article • 11/15/2023

This article discusses how to install .NET on Ubuntu 22.04; .NET 8, .NET 7, and .NET 6, are supported.

Install the SDK (which includes the runtime) if you want to develop .NET apps. Or, if you only need to run apps, install the Runtime. If you're installing the Runtime, we suggest you install the **ASP.NET Core Runtime** as it includes both .NET and ASP.NET Core runtimes.

Use the `dotnet --list-sdks` and `dotnet --list-runtimes` commands to see which versions are installed. For more information, see [How to check that .NET is already installed](#).

ⓘ Important

Package manager installs are only supported on the **x64** architecture. Other architectures, such as **Arm**, must install .NET by some other means such as with Snap, an installer script, or through a manual binary installation.

For more information on installing .NET **without a package manager**, see one of the following articles:

- [Alternatively install .NET with install-dotnet script](#).
- [Manually install .NET](#)

.NET is available in the [Ubuntu packages feed](#), as well as the Microsoft package repository. However, you should only use one or the other to install .NET. If you want to use the Microsoft package repository, see [How to register the Microsoft package repository](#).

Supported versions

The following versions of .NET are supported or available for Ubuntu 22.04:

| Supported .NET versions | Available in Ubuntu feed | Available in Microsoft feed |
|-------------------------|--------------------------|-----------------------------|
| 8.0, 7.0, 6.0 | 7.0, 6.0 | 8.0, 7.0, 6.0, 3.1 |

ⓘ Important

.NET SDK versions offered by Canonical are always in the **.1xx feature band**. If you want to use a newer feature band release, use the [Microsoft feed to install the SDK](#). Make sure you review the information in the [.NET package mix ups on Linux](#) article to understand the implications of switching between repository feeds.

When an [Ubuntu version](#) falls out of support, .NET is no longer supported with that version.

The following versions of .NET are **✗** no longer supported:

- .NET 5
- .NET Core 3.1
- .NET Core 3.0
- .NET Core 2.2
- .NET Core 2.1
- .NET Core 2.0

Install .NET

Install the SDK

The .NET SDK allows you to develop apps with .NET. If you install the .NET SDK, you don't need to install the corresponding runtime. To install the .NET SDK, run the following commands:

Bash

```
sudo apt-get update && \
  sudo apt-get install -y dotnet-sdk-8.0
```

ⓘ Important

If you receive an error message similar to **Unable to locate package dotnet-sdk-8.0**, see the [troubleshooting](#) section.

Install the runtime

The ASP.NET Core Runtime allows you to run apps that were made with .NET that didn't provide the runtime. The following commands install the ASP.NET Core Runtime, which is the most compatible runtime for .NET. In your terminal, run the following commands:

Bash

```
sudo apt-get update && \
  sudo apt-get install -y aspnetcore-runtime-8.0
```

ⓘ Important

If you receive an error message similar to **Unable to locate package aspnetcore-runtime-8.0**, see the [troubleshooting](#) section.

As an alternative to the ASP.NET Core Runtime, you can install the .NET Runtime, which doesn't include ASP.NET Core support: replace `aspnetcore-runtime-8.0` in the previous command with `dotnet-runtime-8.0`:

Bash

```
sudo apt-get install -y dotnet-runtime-8.0
```

How to install other versions

Other versions of .NET aren't supported in the Ubuntu feeds. Instead, use the Microsoft package repository.

Troubleshooting

If you run into issues installing or even running .NET, see [Troubleshooting](#).

Dependencies

When you install with a package manager, these libraries are installed for you. But, if you manually install .NET or you publish a self-contained app, you'll need to make sure these libraries are installed:

- libc6
- libgcc1
- libgcc-s1

- libgssapi-krb5-2
- libicu70
- libltng-ust1
- libssl3
- libstdc++6
- libunwind8
- zlib1g

Dependencies can be installed with the `apt install` command. The following snippet demonstrates installing the `zlib1g` library:

Bash

```
sudo apt install zlib1g
```

If the .NET app uses the `System.Drawing.Common` assembly, `libgdiplus` will also need to be installed. Because [System.Drawing.Common is no longer supported on Linux](#), this only works on .NET 6 and requires setting the `System.Drawing.EnableUnixSupport` runtime configuration switch.

You can install a recent version of `libgdiplus` by [adding the Mono repository to your system ↗](#).

Next steps

- [How to enable TAB completion for the .NET CLI](#)
- [Tutorial: Create a console application with .NET SDK using Visual Studio Code](#)

 Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

Install .NET SDK or .NET Runtime on Ubuntu 20.04

Article • 11/21/2023

This article discusses how to install .NET on Ubuntu 20.04; .NET 8, .NET 7, and .NET 6, are supported.

Install the SDK (which includes the runtime) if you want to develop .NET apps. Or, if you only need to run apps, install the Runtime. If you're installing the Runtime, we suggest you install the **ASP.NET Core Runtime** as it includes both .NET and ASP.NET Core runtimes.

Use the `dotnet --list-sdks` and `dotnet --list-runtimes` commands to see which versions are installed. For more information, see [How to check that .NET is already installed](#).

ⓘ Important

Package manager installs are only supported on the **x64** architecture. Other architectures, such as **Arm**, must install .NET by some other means such as with Snap, an installer script, or through a manual binary installation.

For more information on installing .NET **without a package manager**, see one of the following articles:

- [Alternatively install .NET with install-dotnet script.](#)
- [Manually install .NET](#)

Supported versions

The following versions of .NET are supported or available for Ubuntu 20.04:

| Supported .NET versions | Available in Ubuntu feed | Available in Microsoft feed |
|-------------------------|--------------------------|------------------------------|
| 8.0, 7.0, 6.0 | None | 8.0, 7.0, 6.0, 5.0, 3.1, 2.1 |

When an [Ubuntu version](#) falls out of support, .NET is no longer supported with that version.

The following versions of .NET are  no longer supported:

- .NET 5
- .NET Core 3.1
- .NET Core 3.0
- .NET Core 2.2
- .NET Core 2.1
- .NET Core 2.0

Add the Microsoft package repository

Installing with APT can be done with a few commands. Before you install .NET, run the following commands to add the Microsoft package signing key to your list of trusted keys and add the package repository.

Open a terminal and run the following commands:

Bash

```
wget https://packages.microsoft.com/config/ubuntu/20.04/packages-microsoft-prod.deb -O packages-microsoft-prod.deb  
sudo dpkg -i packages-microsoft-prod.deb  
rm packages-microsoft-prod.deb
```

Install the SDK

The .NET SDK allows you to develop apps with .NET. If you install the .NET SDK, you don't need to install the corresponding runtime. To install the .NET SDK, run the following commands:

Bash

```
sudo apt-get update && \  
sudo apt-get install -y dotnet-sdk-8.0
```

ⓘ Important

If you receive an error message similar to **Unable to locate package dotnet-sdk-8.0**, see the [troubleshooting](#) section.

Install the runtime

The ASP.NET Core Runtime allows you to run apps that were made with .NET that didn't provide the runtime. The following commands install the ASP.NET Core Runtime, which is the most compatible runtime for .NET. In your terminal, run the following commands:

Bash

```
sudo apt-get update && \
sudo apt-get install -y aspnetcore-runtime-8.0
```

ⓘ Important

If you receive an error message similar to **Unable to locate package aspnetcore-runtime-8.0**, see the [troubleshooting](#) section.

As an alternative to the ASP.NET Core Runtime, you can install the .NET Runtime, which doesn't include ASP.NET Core support: replace `aspnetcore-runtime-8.0` in the previous command with `dotnet-runtime-8.0`:

Bash

```
sudo apt-get install -y dotnet-runtime-8.0
```

How to install other versions

All versions of .NET are available for download at

<https://dotnet.microsoft.com/download/dotnet>, but require [manual installation](#). You can try and use the package manager to install a different version of .NET. However, the requested version may not be available.

The packages added to package manager feeds are named in a hackable format, for example: `{product}-{type}-{version}`.

- **product**

The type of .NET product to install. Valid options are:

- `dotnet`
- `aspnetcore`

- **type**

Chooses the SDK or the runtime. Valid options are:

- `sdk` (only available for the `dotnet` product)
- `runtime`

- **version**

The version of the SDK or runtime to install. This article will always give the instructions for the latest supported version. Valid options are any released version, such as:

- 8.0
- 6.0
- 3.1
- 2.1

It's possible the SDK/runtime you're trying to download is not available for your Linux distribution. For a list of supported distributions, see [Install .NET on Linux](#).

Examples

- Install the ASP.NET Core 8.0 runtime: `aspnetcore-runtime-8.0`
- Install the .NET Core 2.1 runtime: `dotnet-runtime-2.1`
- Install the .NET 5 SDK: `dotnet-sdk-5.0`
- Install the .NET Core 3.1 SDK: `dotnet-sdk-3.1`

Package missing

If the package-version combination doesn't work, it's not available. For example, there isn't an ASP.NET Core SDK, the SDK components are included with the .NET SDK. The value `aspnetcore-sdk-8.0` is incorrect and should be `dotnet-sdk-8.0`. For a list of Linux distributions supported by .NET, see [.NET dependencies and requirements](#).

Troubleshooting

If you run into issues installing or even running .NET, see [Troubleshooting](#).

Dependencies

When you install with a package manager, these libraries are installed for you. But, if you manually install .NET or you publish a self-contained app, you'll need to make sure these libraries are installed:

- libc6
- libgcc1
- libgssapi-krb5-2
- libicu66

- libssl1.1
- libstdc++6
- zlib1g

Dependencies can be installed with the `apt install` command. The following snippet demonstrates installing the `zlib1g` library:

Bash

```
sudo apt install zlib1g
```

If the .NET app uses the `System.Drawing.Common` assembly, `libgdiplus` will also need to be installed. Because `System.Drawing.Common` is no longer supported on Linux, this only works on .NET 6 and requires setting the `System.Drawing.EnableUnixSupport` runtime configuration switch.

You can install a recent version of `libgdiplus` by [adding the Mono repository to your system](#).

Next steps

- [How to enable TAB completion for the .NET CLI](#)
- [Tutorial: Create a console application with .NET SDK using Visual Studio Code](#)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

Install .NET SDK or .NET Runtime on Ubuntu 18.04

Article • 08/17/2023

This article discusses how to install .NET on Ubuntu 18.04; .NET 6 and .NET 7 are supported.

Install the SDK (which includes the runtime) if you want to develop .NET apps. Or, if you only need to run apps, install the Runtime. If you're installing the Runtime, we suggest you install the **ASP.NET Core Runtime** as it includes both .NET and ASP.NET Core runtimes.

Use the `dotnet --list-sdks` and `dotnet --list-runtimes` commands to see which versions are installed. For more information, see [How to check that .NET is already installed](#).

ⓘ Important

Package manager installs are only supported on the **x64** architecture. Other architectures, such as **Arm**, must install .NET by some other means such as with Snap, an installer script, or through a manual binary installation.

For more information on installing .NET **without a package manager**, see one of the following articles:

- [Alternatively install .NET with install-dotnet script.](#)
- [Manually install .NET](#)

Supported versions

The following versions of .NET are supported or available for Ubuntu 18.04:

| Supported .NET versions | Available in Ubuntu feed | Available in Microsoft feed |
|-------------------------|--------------------------|------------------------------|
| 7.0, 6.0 | None | 7.0, 6.0, 5.0, 3.1, 2.2, 2.1 |

When an [Ubuntu version](#) falls out of support, .NET is no longer supported with that version.

The following versions of .NET are  no longer supported:

- .NET 5
- .NET Core 3.1
- .NET Core 3.0
- .NET Core 2.2
- .NET Core 2.1
- .NET Core 2.0

Add the Microsoft package repository

Installing with APT can be done with a few commands. Before you install .NET, run the following commands to add the Microsoft package signing key to your list of trusted keys and add the package repository.

Open a terminal and run the following commands:

Bash

```
wget https://packages.microsoft.com/config/ubuntu/18.04/packages-microsoft-prod.deb -O packages-microsoft-prod.deb  
sudo dpkg -i packages-microsoft-prod.deb  
rm packages-microsoft-prod.deb
```

Install the SDK

The .NET SDK allows you to develop apps with .NET. If you install the .NET SDK, you don't need to install the corresponding runtime. To install the .NET SDK, run the following commands:

Bash

```
sudo apt-get update && \  
sudo apt-get install -y dotnet-sdk-7.0
```

ⓘ Important

If you receive an error message similar to **Unable to locate package dotnet-sdk-7.0**, see the [troubleshooting](#) section.

Install the runtime

The ASP.NET Core Runtime allows you to run apps that were made with .NET that didn't provide the runtime. The following commands install the ASP.NET Core Runtime, which is the most compatible runtime for .NET. In your terminal, run the following commands:

Bash

```
sudo apt-get update && \
sudo apt-get install -y aspnetcore-runtime-7.0
```

ⓘ Important

If you receive an error message similar to **Unable to locate package aspnetcore-runtime-7.0**, see the [troubleshooting](#) section.

As an alternative to the ASP.NET Core Runtime, you can install the .NET Runtime, which doesn't include ASP.NET Core support: replace `aspnetcore-runtime-7.0` in the previous command with `dotnet-runtime-7.0`:

Bash

```
sudo apt-get install -y dotnet-runtime-7.0
```

How to install other versions

All versions of .NET are available for download at

<https://dotnet.microsoft.com/download/dotnet>, but require [manual installation](#). You can try and use the package manager to install a different version of .NET. However, the requested version may not be available.

The packages added to package manager feeds are named in a hackable format, for example: `{product}-{type}-{version}`.

- **product**

The type of .NET product to install. Valid options are:

- `dotnet`
- `aspnetcore`

- **type**

Chooses the SDK or the runtime. Valid options are:

- `sdk` (only available for the `dotnet` product)
- `runtime`

- **version**

The version of the SDK or runtime to install. This article will always give the instructions for the latest supported version. Valid options are any released version, such as:

- 8.0
- 6.0
- 3.1
- 2.1

It's possible the SDK/runtime you're trying to download is not available for your Linux distribution. For a list of supported distributions, see [Install .NET on Linux](#).

Examples

- Install the ASP.NET Core 8.0 runtime: `aspnetcore-runtime-8.0`
- Install the .NET Core 2.1 runtime: `dotnet-runtime-2.1`
- Install the .NET 5 SDK: `dotnet-sdk-5.0`
- Install the .NET Core 3.1 SDK: `dotnet-sdk-3.1`

Package missing

If the package-version combination doesn't work, it's not available. For example, there isn't an ASP.NET Core SDK, the SDK components are included with the .NET SDK. The value `aspnetcore-sdk-8.0` is incorrect and should be `dotnet-sdk-8.0`. For a list of Linux distributions supported by .NET, see [.NET dependencies and requirements](#).

Troubleshooting

If you run into issues installing or even running .NET, see [Troubleshooting](#).

Dependencies

When you install with a package manager, these libraries are installed for you. But, if you manually install .NET or you publish a self-contained app, you'll need to make sure these libraries are installed:

- libc6
- libgcc1
- libgssapi-krb5-2
- libicu60

- libssl1.1
- libstdc++6
- zlib1g

Dependencies can be installed with the `apt install` command. The following snippet demonstrates installing the `zlib1g` library:

Bash

```
sudo apt install zlib1g
```

If the .NET app uses the `System.Drawing.Common` assembly, `libgdiplus` will also need to be installed. Because `System.Drawing.Common` is no longer supported on Linux, this only works on .NET 6 and requires setting the `System.Drawing.EnableUnixSupport` runtime configuration switch.

You can install a recent version of `libgdiplus` by [adding the Mono repository to your system](#).

Next steps

- [How to enable TAB completion for the .NET CLI](#)
- [Tutorial: Create a console application with .NET SDK using Visual Studio Code](#)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

Install .NET SDK or .NET Runtime on Ubuntu 16.04

Article • 08/17/2023

This article discusses how to install .NET on Ubuntu 16.04; Only .NET 6 is supported.

Install the SDK (which includes the runtime) if you want to develop .NET apps. Or, if you only need to run apps, install the Runtime. If you're installing the Runtime, we suggest you install the **ASP.NET Core Runtime** as it includes both .NET and ASP.NET Core runtimes.

Use the `dotnet --list-sdks` and `dotnet --list-runtimes` commands to see which versions are installed. For more information, see [How to check that .NET is already installed](#).

ⓘ Important

Package manager installs are only supported on the **x64** architecture. Other architectures, such as **Arm**, must install .NET by some other means such as with Snap, an installer script, or through a manual binary installation.

For more information on installing .NET **without a package manager**, see one of the following articles:

- [Alternatively install .NET with install-dotnet script](#).
- [Manually install .NET](#)

Supported versions

The following versions of .NET are supported or available for Ubuntu 16.04:

| Supported .NET versions | Available in Ubuntu feed | Available in Microsoft feed |
|-------------------------|--------------------------|-----------------------------------|
| 6.0 | None | 6.0, 5.0, 3.1, 3.0, 2.2, 2.1, 2.0 |

When an [Ubuntu version](#) falls out of support, .NET is no longer supported with that version.

The following versions of .NET are  no longer supported:

- .NET 5

- .NET Core 3.1
- .NET Core 3.0
- .NET Core 2.2
- .NET Core 2.1
- .NET Core 2.0

Add the Microsoft package repository

Installing with APT can be done with a few commands. Before you install .NET, run the following commands to add the Microsoft package signing key to your list of trusted keys and add the package repository.

Open a terminal and run the following commands:

Bash

```
wget https://packages.microsoft.com/config/ubuntu/16.04/packages-microsoft-prod.deb -O packages-microsoft-prod.deb  
sudo dpkg -i packages-microsoft-prod.deb  
rm packages-microsoft-prod.deb
```

Install .NET 6 SDK

The .NET SDK allows you to develop apps with .NET. If you install the .NET SDK, you don't need to install the corresponding runtime. To install the .NET SDK, run the following commands:

Bash

```
sudo apt-get update && \  
sudo apt-get install -y dotnet-sdk-6.0
```

Important

If you receive an error message similar to **Unable to locate package dotnet-sdk-6.0**, see the [troubleshooting](#) section.

Install .NET 6 Runtime

The ASP.NET Core Runtime allows you to run apps that were made with .NET that didn't provide the runtime. The following command installs the ASP.NET Core Runtime, which

is the most compatible runtime for .NET. In your terminal, run the following commands:

Bash

```
sudo apt-get update && \
  sudo apt-get install -y aspnetcore-runtime-6.0
```

ⓘ Important

If you receive an error message similar to **Unable to locate package aspnetcore-runtime-6.0**, see the [troubleshooting](#) section.

As an alternative to the ASP.NET Core Runtime, you can install the .NET Runtime, which doesn't include ASP.NET Core support: replace `aspnetcore-runtime-6.0` in the previous command with `dotnet-runtime-6.0`:

Bash

```
sudo apt-get install -y dotnet-runtime-6.0
```

How to install other versions

All versions of .NET are available for download at <https://dotnet.microsoft.com/download/dotnet>, but require [manual installation](#). You can try and use the package manager to install a different version of .NET. However, the requested version may not be available.

The packages added to package manager feeds are named in a hackable format, for example: `{product}-{type}-{version}`.

- **product**

The type of .NET product to install. Valid options are:

- `dotnet`
- `aspnetcore`

- **type**

Chooses the SDK or the runtime. Valid options are:

- `sdk` (only available for the `dotnet` product)
- `runtime`

- **version**

The version of the SDK or runtime to install. This article will always give the instructions for the latest supported version. Valid options are any released version, such as:

- 8.0
- 6.0
- 3.1
- 2.1

It's possible the SDK/runtime you're trying to download is not available for your Linux distribution. For a list of supported distributions, see [Install .NET on Linux](#).

Examples

- Install the ASP.NET Core 8.0 runtime: `aspnetcore-runtime-8.0`
- Install the .NET Core 2.1 runtime: `dotnet-runtime-2.1`
- Install the .NET 5 SDK: `dotnet-sdk-5.0`
- Install the .NET Core 3.1 SDK: `dotnet-sdk-3.1`

Package missing

If the package-version combination doesn't work, it's not available. For example, there isn't an ASP.NET Core SDK, the SDK components are included with the .NET SDK. The value `aspnetcore-sdk-8.0` is incorrect and should be `dotnet-sdk-8.0`. For a list of Linux distributions supported by .NET, see [.NET dependencies and requirements](#).

Troubleshooting

If you run into issues installing or even running .NET, see [Troubleshooting](#).

Dependencies

When you install with a package manager, these libraries are installed for you. But, if you manually install .NET or you publish a self-contained app, you'll need to make sure these libraries are installed:

- libc6
- libgcc1
- libgssapi-krb5-2
- libicu55

- libssl1.0.0
- libstdc++6
- zlib1g

Dependencies can be installed with the `apt install` command. The following snippet demonstrates installing the `zlib1g` library:

Bash

```
sudo apt install zlib1g
```

If the .NET app uses the `System.Drawing.Common` assembly, `libgdiplus` will also need to be installed. Because `System.Drawing.Common` is no longer supported on Linux, this only works on .NET 6 and requires setting the `System.Drawing.EnableUnixSupport` runtime configuration switch.

You can install a recent version of `libgdiplus` by [adding the Mono repository to your system](#).

Next steps

- [How to enable TAB completion for the .NET CLI](#)
- [Tutorial: Create a console application with .NET SDK using Visual Studio Code](#)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

Install the .NET SDK or the .NET Runtime on Alpine

Article • 01/10/2024

ⓘ Important

.NET 8 was released on November 14, 2023. It may take time for the packages to appear in the package manager feeds.

.NET is supported on Alpine and this article describes how to install .NET on Alpine. When an Alpine version falls out of support, .NET is no longer supported with that version.

If you're using Docker, consider using [official .NET Docker images](#) instead of installing .NET yourself.

Install the SDK (which includes the runtime) if you want to develop .NET apps. Or, if you only need to run apps, install the Runtime. If you're installing the Runtime, we suggest you install the **ASP.NET Core Runtime** as it includes both .NET and ASP.NET Core runtimes.

Use the `dotnet --list-sdks` and `dotnet --list-runtimes` commands to see which versions are installed. For more information, see [How to check that .NET is already installed](#).

The Alpine package manager supports installing some versions of .NET. If the .NET package is unavailable, you'll need to install .NET in one of the following alternative ways:

- [Use the .NET install script.](#)
- [Download and install .NET manually.](#)

Install .NET 8

.NET 8 isn't yet available in the official Alpine package repository. Use one of the following ways to install .NET 8:

- [Use the .NET install script.](#)
- [Download and install .NET manually.](#)

Install .NET 7

Install the SDK

The .NET SDK allows you to develop apps with .NET. If you install the .NET SDK, you don't need to install the corresponding runtime. To install the .NET SDK, run the following command:

Bash

```
sudo apk add dotnet7-sdk
```

Install the runtime

The ASP.NET Core Runtime allows you to run apps that were made with .NET that didn't provide the runtime. The following command installs the ASP.NET Core Runtime, which is the most compatible runtime for .NET. In your terminal, run the following command:

Bash

```
sudo apk add aspnetcore7-runtime
```

As an alternative to the ASP.NET Core Runtime, you can install the .NET Runtime, which doesn't include ASP.NET Core support: replace `aspnetcore7-runtime` in the previous command with `dotnet7-runtime`:

Bash

```
sudo apk add dotnet7-runtime
```

Install .NET 6

Install the SDK

The .NET SDK allows you to develop apps with .NET. If you install the .NET SDK, you don't need to install the corresponding runtime. To install the .NET SDK, run the following command:

Bash

```
sudo apk add dotnet6-sdk
```

Install the runtime

The ASP.NET Core Runtime allows you to run apps that were made with .NET that didn't provide the runtime. The following command installs the ASP.NET Core Runtime, which is the most compatible runtime for .NET. In your terminal, run the following command:

Bash

```
sudo apk add aspnetcore6-runtime
```

As an alternative to the ASP.NET Core Runtime, you can install the .NET Runtime, which doesn't include ASP.NET Core support. To install it, replace `aspnetcore6-runtime` in the previous command with `dotnet6-runtime`:

Bash

```
sudo apk add dotnet6-runtime
```

Supported distributions

The following table is a list of currently supported .NET releases and the versions of Alpine they're supported on. These versions remain supported until either the version of [.NET reaches end-of-support](#) or the version of [Alpine reaches end-of-life](#).

[] Expand table

| Alpine | Supported Version | Available in Package Manager |
|--------|--------------------|------------------------------|
| 3.18 | .NET 7.0, .NET 6.0 | .NET 8.0, .NET 7.0, .NET 6.0 |
| 3.17 | .NET 7.0, .NET 6.0 | .NET 8.0, .NET 7.0, .NET 6.0 |
| 3.16 | .NET 7.0, .NET 6.0 | .NET 6.0 |
| 3.15 | .NET 7.0, .NET 6.0 | None |

The following versions of .NET are X no longer supported:

- .NET 5
- .NET Core 3.1

- .NET Core 3.0
- .NET Core 2.2
- .NET Core 2.1
- .NET Core 2.0

Supported architectures

The following table is a list of currently supported .NET releases and the architecture of Alpine they're supported on. These versions remain supported until either the version of [.NET reaches end-of-support](#) or the architecture of [Alpine is supported](#). Note that only `x86_64`, `armv7`, `aarch64` is officially supported by Microsoft. Other architectures are supported by the distribution maintainers, and can be installed using the `apk` package manager.

[\[\] Expand table](#)

| Architecture | .NET 6 | .NET 7 | .NET 8 |
|--------------|------------------|------------|------------|
| x86_64 | 3.16, 3.17, 3.18 | 3.17, 3.18 | 3.17, 3.18 |
| x86 | None | None | None |
| aarch64 | 3.16, 3.17, 3.18 | 3.17, 3.18 | 3.17, 3.18 |
| armv7 | 3.16, 3.17, 3.18 | 3.17, 3.18 | 3.17, 3.18 |
| armhf | None | None | None |
| s390x | 3.17 | 3.17 | 3.17 |
| ppc64le | None | None | None |
| riscv64 | None | None | None |

Install preview versions

Preview and release candidate versions of .NET aren't available in package repositories. You can install previews and release candidates of .NET in one of the following ways:

- [Scripted install with `install-dotnet.sh`](#)
- [Manual binary extraction](#)

Remove preview versions

When using a package manager to manage your installation of .NET, you may run into a conflict if you've previously installed a preview release. The package manager may interpret the non-preview release as an earlier version of .NET. To install the non-preview release, first uninstall the preview versions. For more information about uninstalling .NET, see [How to remove the .NET Runtime and SDK](#).

Dependencies

When you install with a package manager, these libraries are installed for you. But, if you manually install .NET or you publish a self-contained app, you'll need to make sure these libraries are installed:

- icu-libs
- krb5-libs
- libgcc
- libgdiplus (if the .NET app requires the *System.Drawing.Common* assembly)
- libintl
- libssl1.1 (for 3.14.x and older)
- libssl3 (for 3.15.x and newer)
- libstdc++
- zlib

To install the needed requirements, run the following command:

```
Bash
```

```
apk add bash icu-libs krb5-libs libgcc libintl libssl1.1 libstdc++ zlib
```

If the .NET app uses the *System.Drawing.Common* assembly, libgdiplus will also need to be installed. Because [*System.Drawing.Common* is no longer supported on Linux](#), this only works on .NET 6 and requires setting the `System.Drawing.EnableUnixSupport` runtime configuration switch.

To install libgdiplus on Alpine 3.16 or later, run:

```
Bash
```

```
apk add libgdiplus
```

Next steps

- How to enable TAB completion for the .NET CLI
- Tutorial: Create a console application with .NET SDK using Visual Studio Code

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

.NET is an open source project.
Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

Install the .NET SDK or the .NET Runtime on CentOS Linux

Article • 05/04/2023

.NET is supported on CentOS Linux. This article describes how to install .NET on CentOS Linux. If you need to install .NET On CentOS Stream, see [Install the .NET SDK or the .NET Runtime on RHEL and CentOS Stream](#).

Install the SDK (which includes the runtime) if you want to develop .NET apps. Or, if you only need to run apps, install the Runtime. If you're installing the Runtime, we suggest you install the **ASP.NET Core Runtime** as it includes both .NET and ASP.NET Core runtimes.

Use the `dotnet --list-sdks` and `dotnet --list-runtimes` commands to see which versions are installed. For more information, see [How to check that .NET is already installed](#).

Important

Package manager installs are only supported on the **x64** architecture. Other architectures, such as **Arm**, must install .NET by some other means such as with Snap, an installer script, or through a manual binary installation.

For more information on installing .NET **without a package manager**, see one of the following articles:

- [Alternatively install .NET with install-dotnet script](#).
- [Manually install .NET](#)

Supported distributions

The following table is a list of currently supported .NET releases on CentOS Linux 7. These versions remain supported until either the version of .NET reaches [end-of-support](#) or the version of CentOS Linux is no longer supported.

| CentOS Linux | .NET |
|--------------|------|
| 7 | 7, 6 |

Warning

CentOS Linux 8 reached an early End Of Life (EOL) on December 31st, 2021. For more information, see the official [CentOS Linux EOL page](#). Because of this, .NET isn't supported on CentOS Linux 8.

The following versions of .NET are  no longer supported:

- .NET 5
- .NET Core 3.1
- .NET Core 3.0
- .NET Core 2.2
- .NET Core 2.1
- .NET Core 2.0

Important

Package manager installs are only supported on the **x64** architecture. Other architectures, such as **Arm**, must install .NET by some other means such as with Snap, an installer script, or through a manual binary installation.

For more information on installing .NET **without a package manager**, see one of the following articles:

- [Alternatively install .NET with *install-dotnet* script.](#)
- [Manually install .NET](#)

Install preview versions

Preview and release candidate versions of .NET aren't available in package repositories. You can install previews and release candidates of .NET in one of the following ways:

- [Scripted install with *install-dotnet.sh*](#)
- [Manual binary extraction](#)

Remove preview versions

When using a package manager to manage your installation of .NET, you may run into a conflict if you've previously installed a preview release. The package manager may interpret the non-preview release as an earlier version of .NET. To install the non-

preview release, first uninstall the preview versions. For more information about uninstalling .NET, see [How to remove the .NET Runtime and SDK](#).

CentOS Linux 7

Before you install .NET, run the following commands to add the Microsoft package signing key to your list of trusted keys and add the Microsoft package repository. Open a terminal and run the following commands:

Bash

```
sudo rpm -Uvh https://packages.microsoft.com/config/centos/7/packages-microsoft-prod.rpm
```

Install the SDK

The .NET SDK allows you to develop apps with .NET. If you install the .NET SDK, you don't need to install the corresponding runtime. To install the .NET SDK, run the following command:

Bash

```
sudo yum install dotnet-sdk-7.0
```

Install the runtime

The ASP.NET Core Runtime allows you to run apps that were made with .NET that didn't provide the runtime. The following command installs the ASP.NET Core Runtime, which is the most compatible runtime for .NET. In your terminal, run the following command:

Bash

```
sudo yum install aspnetcore-runtime-7.0
```

As an alternative to the ASP.NET Core Runtime, you can install the .NET Runtime, which doesn't include ASP.NET Core support: replace `aspnetcore-runtime-7.0` in the previous command with `dotnet-runtime-7.0`:

Bash

```
sudo yum install dotnet-runtime-7.0
```

How to install other versions

All versions of .NET are available for download at <https://dotnet.microsoft.com/download/dotnet>, but require [manual installation](#). You can try and use the package manager to install a different version of .NET. However, the requested version may not be available.

The packages added to package manager feeds are named in a hackable format, for example: `{product}-{type}-{version}`.

- **product**

The type of .NET product to install. Valid options are:

- dotnet
- aspnetcore

- **type**

Chooses the SDK or the runtime. Valid options are:

- sdk (only available for the **dotnet** product)
- runtime

- **version**

The version of the SDK or runtime to install. This article will always give the instructions for the latest supported version. Valid options are any released version, such as:

- 8.0
- 6.0
- 3.1
- 2.1

It's possible the SDK/runtime you're trying to download is not available for your Linux distribution. For a list of supported distributions, see [Install .NET on Linux](#).

Examples

- Install the ASP.NET Core 8.0 runtime: `aspnetcore-runtime-8.0`
- Install the .NET Core 2.1 runtime: `dotnet-runtime-2.1`
- Install the .NET 5 SDK: `dotnet-sdk-5.0`
- Install the .NET Core 3.1 SDK: `dotnet-sdk-3.1`

Package missing

If the package-version combination doesn't work, it's not available. For example, there isn't an ASP.NET Core SDK, the SDK components are included with the .NET SDK. The value `aspnetcore-sdk-8.0` is incorrect and should be `dotnet-sdk-8.0`. For a list of Linux distributions supported by .NET, see [.NET dependencies and requirements](#).

Troubleshoot the package manager

This section provides information on common errors you may get while using the package manager to install .NET.

Unable to find package

Important

Package manager installs are only supported on the **x64** architecture. Other architectures, such as **Arm**, must install .NET by some other means such as with Snap, an installer script, or through a manual binary installation.

For more information on installing .NET **without a package manager**, see one of the following articles:

- [Alternatively install .NET with install-dotnet script](#).
- [Manually install .NET](#)

Failed to fetch

While installing the .NET package, you may see an error similar to `signature verification failed for file 'repomd.xml' from repository 'packages-microsoft-com-prod'`. Generally speaking, this error means that the package feed for .NET is being upgraded with newer package versions, and that you should try again later. During an upgrade, the package feed should not be unavailable for more than 2 hours. If you continually receive this error for more than 2 hours, please file an issue at <https://github.com/dotnet/core/issues>.

Errors related to missing `fxr`, `libhostfxr.so`, or `FrameworkList.xml`

For more information about solving these problems, see [Troubleshoot fxr, libhostfxr.so, and FrameworkList.xml errors](#).

Dependencies

When you install with a package manager, these libraries are installed for you. But, if you manually install .NET or you publish a self-contained app, you'll need to make sure these libraries are installed:

- krb5-libs
- libicu
- openssl-libs
- zlib

If the target runtime environment's OpenSSL version is 1.1 or newer, you'll need to install `compat-openssl110`.

Dependencies can be installed with the `yum install` command. The following snippet demonstrates installing the `libicu` library:

```
Bash
```

```
sudo yum install libicu
```

For more information about the dependencies, see [Self-contained Linux apps ↗](#).

If the .NET app uses the `System.Drawing.Common` assembly, `libgdiplus` will also need to be installed. Because [System.Drawing.Common is no longer supported on Linux](#), this only works on .NET 6 and requires setting the `System.Drawing.EnableUnixSupport` runtime configuration switch.

You can install a recent version of `libgdiplus` by [adding the Mono repository to your system ↗](#).

Next steps

- [How to enable TAB completion for the .NET CLI](#)
- [Tutorial: Create a console application with .NET SDK using Visual Studio Code](#)



Collaborate with us on
GitHub

The source for this content can
be found on GitHub, where you



.NET feedback

The .NET documentation is open
source. Provide feedback here.

can also create and review issues and pull requests. For more information, see [our contributor guide](#).

 [Open a documentation issue](#)

 [Provide product feedback](#)

Install the .NET SDK or the .NET Runtime on RHEL and CentOS Stream

Article • 11/15/2023

ⓘ Important

.NET 8 was released on November 14, 2023. It may take time for the packages to appear in the package manager feeds.

.NET is supported on Red Hat Enterprise Linux (RHEL). This article describes how to install .NET on RHEL and CentOS Stream.

Install the SDK (which includes the runtime) if you want to develop .NET apps. Or, if you only need to run apps, install the Runtime. If you're installing the Runtime, we suggest you install the **ASP.NET Core Runtime** as it includes both .NET and ASP.NET Core runtimes.

Use the `dotnet --list-sdks` and `dotnet --list-runtimes` commands to see which versions are installed. For more information, see [How to check that .NET is already installed](#).

Register your Red Hat subscription

To install .NET from Red Hat on RHEL, you first need to register using the Red Hat Subscription Manager. If this hasn't been done on your system, or if you're unsure, see the [Red Hat Product Documentation for .NET](#).

ⓘ Important

This doesn't apply to CentOS Stream.

Supported distributions

The following table is a list of currently supported .NET releases on both RHEL and CentOS Stream. These versions remain supported until either the version of [.NET reaches end-of-support](#) or the Linux distribution is no longer supported.

| Distribution | .NET |
|-----------------|---------|
| RHEL 9 (9.1) | 8, 7, 6 |
| RHEL 8 (8.7) | 8, 7, 6 |
| RHEL 7 | 6 |
| CentOS Stream 9 | 8, 7, 6 |
| CentOS Stream 8 | 8, 7, 6 |

The following versions of .NET are  no longer supported:

- .NET 5
- .NET Core 3.1
- .NET Core 3.0
- .NET Core 2.2
- .NET Core 2.1
- .NET Core 2.0

Install preview versions

Preview and release candidate versions of .NET aren't available in package repositories. You can install previews and release candidates of .NET in one of the following ways:

- Scripted install with [*install-dotnet.sh*](#)
- Manual binary extraction

Remove preview versions

When using a package manager to manage your installation of .NET, you may run into a conflict if you've previously installed a preview release. The package manager may interpret the non-preview release as an earlier version of .NET. To install the non-preview release, first uninstall the preview versions. For more information about uninstalling .NET, see [How to remove the .NET Runtime and SDK](#).

RHEL 9

.NET is included in the AppStream repositories for RHEL 9.

Important

.NET 8 was released on November 14, 2023. It may take time for the packages to appear in the package manager feeds.

Install the SDK

The .NET SDK allows you to develop apps with .NET. If you install the .NET SDK, you don't need to install the corresponding runtime. To install the .NET SDK, run the following command:

Bash

```
sudo dnf install dotnet-sdk-8.0
```

Install the runtime

The ASP.NET Core Runtime allows you to run apps that were made with .NET that didn't provide the runtime. The following command installs the ASP.NET Core Runtime, which is the most compatible runtime for .NET. In your terminal, run the following command:

Bash

```
sudo dnf install aspnetcore-runtime-8.0
```

As an alternative to the ASP.NET Core Runtime, you can install the .NET Runtime, which doesn't include ASP.NET Core support: replace `aspnetcore-runtime-8.0` in the previous command with `dotnet-runtime-8.0`:

Bash

```
sudo dnf install dotnet-runtime-8.0
```

RHEL 8

.NET is included in the AppStream repositories for RHEL 8.

Important

.NET 8 was released on November 14, 2023. It may take time for the packages to appear in the package manager feeds.

Install the SDK

The .NET SDK allows you to develop apps with .NET. If you install the .NET SDK, you don't need to install the corresponding runtime. To install the .NET SDK, run the following command:

Bash

```
sudo dnf install dotnet-sdk-8.0
```

Install the runtime

The ASP.NET Core Runtime allows you to run apps that were made with .NET that didn't provide the runtime. The following command installs the ASP.NET Core Runtime, which is the most compatible runtime for .NET. In your terminal, run the following command:

Bash

```
sudo dnf install aspnetcore-runtime-8.0
```

As an alternative to the ASP.NET Core Runtime, you can install the .NET Runtime, which doesn't include ASP.NET Core support: replace `aspnetcore-runtime-8.0` in the previous command with `dotnet-runtime-8.0`:

Bash

```
sudo dnf install dotnet-runtime-8.0
```

RHEL 7 ✗ .NET 8

.NET 8 isn't compatible with RHEL 7 and doesn't work.

RHEL 7 ✗ .NET 7

.NET 7 isn't officially supported on RHEL 7. To install .NET 7, see [Install .NET on Linux by using an install script or by extracting binaries](#).

RHEL 7 ✓ .NET 6

The following command installs the `scl-utils` package:

Bash

```
sudo yum install scl-utils
```

Install the SDK

The .NET SDK allows you to develop apps with .NET. If you install the .NET SDK, you don't need to install the corresponding runtime. To install .NET SDK, run the following commands:

Bash

```
subscription-manager repos --enable=rhel-7-server-dotnet-rpms
yum install rh-dotnet60 -y
scl enable rh-dotnet60 bash
```

Red Hat does not recommend permanently enabling `rh-dotnet60` because it may affect other programs. If you want to enable `rh-dotnet` permanently, add the following line to your `~/.bashrc` file.

Bash

```
source scl_source enable rh-dotnet60
```

Install the runtime

The .NET Runtime allows you to run apps that were made with .NET that didn't include the runtime. The commands below install the ASP.NET Core Runtime, which is the most compatible runtime for .NET Core. In your terminal, run the following commands.

Bash

```
subscription-manager repos --enable=rhel-7-server-dotnet-rpms
yum install rh-dotnet60-aspnetcore-runtime-6.0 -y
scl enable rh-dotnet60 bash
```

Red Hat does not recommend permanently enabling `rh-dotnet60` because it may affect other programs. If you want to enable `rh-dotnet60` permanently, add the following line

to your `~/.bashrc` file.

Bash

```
source scl_source enable rh-dotnet60
```

As an alternative to the ASP.NET Core Runtime, you can install the .NET Runtime that doesn't include ASP.NET Core support: replace `rh-dotnet60-aspnetcore-runtime-6.0` in the preceding command with `rh-dotnet60-dotnet-runtime-6.0`.

CentOS Stream 9 ✓

.NET is included in the AppStream repositories for CentOS Stream 9.

Install the SDK

The .NET SDK allows you to develop apps with .NET. If you install the .NET SDK, you don't need to install the corresponding runtime. To install the .NET SDK, run the following command:

Bash

```
sudo dnf install dotnet-sdk-8.0
```

Install the runtime

The ASP.NET Core Runtime allows you to run apps that were made with .NET that didn't provide the runtime. The following command installs the ASP.NET Core Runtime, which is the most compatible runtime for .NET. In your terminal, run the following command:

Bash

```
sudo dnf install aspnetcore-runtime-8.0
```

As an alternative to the ASP.NET Core Runtime, you can install the .NET Runtime, which doesn't include ASP.NET Core support: replace `aspnetcore-runtime-8.0` in the previous command with `dotnet-runtime-8.0`:

Bash

```
sudo dnf install dotnet-runtime-8.0
```

CentOS Stream 8 ✓

Use the Microsoft repository to install .NET:

Bash

```
sudo rpm -Uvh https://packages.microsoft.com/config/centos/8/packages-microsoft-prod.rpm
sudo yum install dotnet-sdk-8.0
```

Dependencies

When you install with a package manager, these libraries are installed for you. But, if you manually install .NET or you publish a self-contained app, you'll need to make sure these libraries are installed:

- krb5-libs
- libicu
- openssl-libs
- zlib

If the target runtime environment's OpenSSL version is 1.1 or newer, you'll need to install `compat-openssl110`.

Dependencies can be installed with the `yum install` command. The following snippet demonstrates installing the `libicu` library:

Bash

```
sudo yum install libicu
```

For more information about the dependencies, see [Self-contained Linux apps ↗](#).

If the .NET app uses the `System.Drawing.Common` assembly, `libgdiplus` will also need to be installed. Because [System.Drawing.Common is no longer supported on Linux](#), this only works on .NET 6 and requires setting the `System.Drawing.EnableUnixSupport` runtime configuration switch.

You can install a recent version of *libgdiplus* by [adding the Mono repository to your system](#).

How to install other versions

Consult the [Red Hat documentation for .NET](#) on the steps required to install other releases of .NET.

Troubleshoot the package manager

This section provides information on common errors you may get while using the package manager to install .NET or .NET Core.

Errors related to missing `fxr`, `libhostfxr.so`, or `FrameworkList.xml`

For more information about solving these problems, see [Troubleshoot fxr, libhostfxr.so, and FrameworkList.xml errors](#).

Next steps

- [How to enable TAB completion for the .NET CLI](#)
- [Tutorial: Create a console application with .NET SDK using Visual Studio Code](#)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



.NET feedback

.NET is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

Install the .NET SDK or the .NET Runtime on Debian

Article • 11/14/2023

This article describes how to install .NET on Debian. When a Debian version falls out of support, .NET is no longer supported with that version. However, these instructions may help you to get .NET running on those versions, even though it isn't supported.

Install the SDK (which includes the runtime) if you want to develop .NET apps. Or, if you only need to run apps, install the Runtime. If you're installing the Runtime, we suggest you install the **ASP.NET Core Runtime** as it includes both .NET and ASP.NET Core runtimes.

Use the `dotnet --list-sdks` and `dotnet --list-runtimes` commands to see which versions are installed. For more information, see [How to check that .NET is already installed](#).

Important

Package manager installs are only supported on the **x64** architecture. Other architectures, such as **Arm**, must install .NET by some other means such as with Snap, an installer script, or through a manual binary installation.

For more information on installing .NET **without a package manager**, see one of the following articles:

- [Alternatively install .NET with install-dotnet script](#).
- [Manually install .NET](#)

Supported distributions

The following table is a list of currently supported .NET releases and the versions of Debian they're supported on. These versions remain supported until either the version of [.NET reaches end-of-support](#) or the version of [Debian reaches end-of-life](#).

| Debian | .NET |
|--------|---------|
| 12 | 8, 7, 6 |
| 11 | 8, 7, 6 |

| Debian | .NET |
|--------|------|
| 10 | 7, 6 |

The following versions of .NET are  no longer supported:

- .NET 5
- .NET Core 3.1
- .NET Core 3.0
- .NET Core 2.2
- .NET Core 2.1
- .NET Core 2.0

Install preview versions

Preview and release candidate versions of .NET aren't available in package repositories. You can install previews and release candidates of .NET in one of the following ways:

- [Scripted install with *install-dotnet.sh*](#)
- [Manual binary extraction](#)

Remove preview versions

When using a package manager to manage your installation of .NET, you may run into a conflict if you've previously installed a preview release. The package manager may interpret the non-preview release as an earlier version of .NET. To install the non-preview release, first uninstall the preview versions. For more information about uninstalling .NET, see [How to remove the .NET Runtime and SDK](#).

Debian 12

Installing with APT can be done with a few commands. Before you install .NET, run the following commands to add the Microsoft package signing key to your list of trusted keys and add the package repository.

Open a terminal and run the following commands:

Bash

```
wget https://packages.microsoft.com/config/debian/12/packages-microsoft-prod.deb -O packages-microsoft-prod.deb
```

```
sudo dpkg -i packages-microsoft-prod.deb  
rm packages-microsoft-prod.deb
```

Install the SDK

The .NET SDK allows you to develop apps with .NET. If you install the .NET SDK, you don't need to install the corresponding runtime. To install the .NET SDK, run the following commands:

Bash

```
sudo apt-get update && \  
sudo apt-get install -y dotnet-sdk-8.0
```

Important

If you receive an error message similar to **Unable to locate package dotnet-sdk-8.0**, see the [troubleshooting](#) section.

Install the runtime

The ASP.NET Core Runtime allows you to run apps that were made with .NET that didn't provide the runtime. The following commands install the ASP.NET Core Runtime, which is the most compatible runtime for .NET. In your terminal, run the following commands:

Bash

```
sudo apt-get update && \  
sudo apt-get install -y aspnetcore-runtime-8.0
```

Important

If you receive an error message similar to **Unable to locate package aspnetcore-runtime-8.0**, see the [troubleshooting](#) section.

As an alternative to the ASP.NET Core Runtime, you can install the .NET Runtime, which doesn't include ASP.NET Core support: replace `aspnetcore-runtime-8.0` in the previous command with `dotnet-runtime-8.0`:

Bash

```
sudo apt-get install -y dotnet-runtime-8.0
```

Debian 11

Installing with APT can be done with a few commands. Before you install .NET, run the following commands to add the Microsoft package signing key to your list of trusted keys and add the package repository.

Open a terminal and run the following commands:

Bash

```
wget https://packages.microsoft.com/config/debian/11/packages-microsoft-prod.deb -O packages-microsoft-prod.deb  
sudo dpkg -i packages-microsoft-prod.deb  
rm packages-microsoft-prod.deb
```

Install the SDK

The .NET SDK allows you to develop apps with .NET. If you install the .NET SDK, you don't need to install the corresponding runtime. To install the .NET SDK, run the following commands:

Bash

```
sudo apt-get update && \  
sudo apt-get install -y dotnet-sdk-8.0
```

ⓘ Important

If you receive an error message similar to **Unable to locate package dotnet-sdk-8.0**, see the [troubleshooting](#) section.

Install the runtime

The ASP.NET Core Runtime allows you to run apps that were made with .NET that didn't provide the runtime. The following commands install the ASP.NET Core Runtime, which is the most compatible runtime for .NET. In your terminal, run the following commands:

Bash

```
sudo apt-get update && \
sudo apt-get install -y aspnetcore-runtime-8.0
```

ⓘ Important

If you receive an error message similar to **Unable to locate package aspnetcore-runtime-8.0**, see the [troubleshooting](#) section.

As an alternative to the ASP.NET Core Runtime, you can install the .NET Runtime, which doesn't include ASP.NET Core support: replace `aspnetcore-runtime-8.0` in the previous command with `dotnet-runtime-8.0`:

Bash

```
sudo apt-get install -y dotnet-runtime-8.0
```

Debian 10

Installing with APT can be done with a few commands. Before you install .NET, run the following commands to add the Microsoft package signing key to your list of trusted keys and add the package repository.

Open a terminal and run the following commands:

Bash

```
wget https://packages.microsoft.com/config/debian/10/packages-microsoft-
prod.deb -O packages-microsoft-prod.deb
sudo dpkg -i packages-microsoft-prod.deb
rm packages-microsoft-prod.deb
```

Install the SDK

The .NET SDK allows you to develop apps with .NET. If you install the .NET SDK, you don't need to install the corresponding runtime. To install the .NET SDK, run the following commands:

Bash

```
sudo apt-get update && \
```

```
sudo apt-get install -y dotnet-sdk-7.0
```

ⓘ Important

If you receive an error message similar to **Unable to locate package dotnet-sdk-7.0**, see the [troubleshooting](#) section.

Install the runtime

The ASP.NET Core Runtime allows you to run apps that were made with .NET that didn't provide the runtime. The following commands install the ASP.NET Core Runtime, which is the most compatible runtime for .NET. In your terminal, run the following commands:

Bash

```
sudo apt-get update && \
  sudo apt-get install -y aspnetcore-runtime-7.0
```

ⓘ Important

If you receive an error message similar to **Unable to locate package aspnetcore-runtime-7.0**, see the [troubleshooting](#) section.

As an alternative to the ASP.NET Core Runtime, you can install the .NET Runtime, which doesn't include ASP.NET Core support: replace `aspnetcore-runtime-7.0` in the previous command with `dotnet-runtime-7.0`:

Bash

```
sudo apt-get install -y dotnet-runtime-7.0
```

How to install other versions

All versions of .NET are available for download at <https://dotnet.microsoft.com/download/dotnet>, but require [manual installation](#). You can try and use the package manager to install a different version of .NET. However, the requested version may not be available.

The packages added to package manager feeds are named in a hackable format, for example: `{product}-{type}-{version}`.

- **product**

The type of .NET product to install. Valid options are:

- dotnet
- aspnetcore

- **type**

Chooses the SDK or the runtime. Valid options are:

- sdk (only available for the **dotnet** product)
- runtime

- **version**

The version of the SDK or runtime to install. This article will always give the instructions for the latest supported version. Valid options are any released version, such as:

- 8.0
- 6.0
- 3.1
- 2.1

It's possible the SDK/runtime you're trying to download is not available for your Linux distribution. For a list of supported distributions, see [Install .NET on Linux](#).

Examples

- Install the ASP.NET Core 8.0 runtime: `aspnetcore-runtime-8.0`
- Install the .NET Core 2.1 runtime: `dotnet-runtime-2.1`
- Install the .NET 5 SDK: `dotnet-sdk-5.0`
- Install the .NET Core 3.1 SDK: `dotnet-sdk-3.1`

Package missing

If the package-version combination doesn't work, it's not available. For example, there isn't an ASP.NET Core SDK, the SDK components are included with the .NET SDK. The value `aspnetcore-sdk-8.0` is incorrect and should be `dotnet-sdk-8.0`. For a list of Linux distributions supported by .NET, see [.NET dependencies and requirements](#).

Use APT to update .NET

When a new patch release is available for .NET, you can simply upgrade it through APT with the following commands:

Bash

```
sudo apt-get update  
sudo apt-get upgrade
```

If you've upgraded your Linux distribution since installing .NET, you may need to reconfigure the Microsoft package repository. Run the installation instructions for your current distribution version to upgrade to the appropriate package repository for .NET updates.

Troubleshooting

This section provides information on common errors you may get while using APT to install .NET.

Unable to find package

(i) Important

Package manager installs are only supported on the **x64** architecture. Other architectures, such as **Arm**, must install .NET by some other means such as with Snap, an installer script, or through a manual binary installation.

For more information on installing .NET **without a package manager**, see one of the following articles:

- [Alternatively install .NET with install-dotnet script.](#)
- [Manually install .NET](#)

Unable to locate \ Some packages could not be installed

If you receive an error message similar to **Unable to locate package {dotnet-package}** or **Some packages could not be installed**, run the following commands.

There are two placeholders in the following set of commands.

- `{dotnet-package}`

This represents the .NET package you're installing, such as `aspnetcore-runtime-`

8.0. This is used in the following `sudo apt-get install` command.

- `{os-version}`

This represents the distribution version you're on. This is used in the `wget` command below. The distribution version is the numerical value, such as `20.04` on Ubuntu or `10` on Debian.

First, try purging the package list:

Bash

```
sudo dpkg --purge packages-microsoft-prod && sudo dpkg -i packages-microsoft-prod.deb  
sudo apt-get update
```

Then, try to install .NET again. If that doesn't work, you can run a manual install with the following commands:

Bash

```
sudo apt-get install -y gpg  
wget -O - https://packages.microsoft.com/keys/microsoft.asc | gpg --dearmor  
-o microsoft.asc.gpg  
sudo mv microsoft.asc.gpg /etc/apt/trusted.gpg.d/  
wget https://packages.microsoft.com/config/debian/{os-version}/prod.list  
sudo mv prod.list /etc/apt/sources.list.d/microsoft-prod.list  
sudo chown root:root /etc/apt/trusted.gpg.d/microsoft.asc.gpg  
sudo chown root:root /etc/apt/sources.list.d/microsoft-prod.list  
sudo apt-get update && \  
    sudo apt-get install -y {dotnet-package}
```

Failed to fetch

While installing the .NET package, you may see an error similar to `Failed to fetch ...`

`File has unexpected size ... Mirror sync in progress?`. This error could mean that the package feed for .NET is being upgraded with newer package versions, and that you should try again later. During an upgrade, the package feed shouldn't be unavailable for more than 30 minutes. If you continually receive this error for more than 30 minutes, please file an issue at <https://github.com/dotnet/core/issues>.

Dependencies

When you install with a package manager, these libraries are installed for you. But, if you manually install .NET or you publish a self-contained app, you'll need to make sure these

libraries are installed:

- libc6
- libgcc-s1
- libgssapi-krb5-2
- libicu63 (for 10.x)
- libicu67 (for 11.x)
- libicu72 (for 12.x)
- libssl1.1
- libstdc++6
- zlib1g

Dependencies can be installed with the `apt install` command. The following snippet demonstrates installing the `libc6` library:

Bash

```
sudo apt install libc6
```

If the .NET app uses the `System.Drawing.Common` assembly, `libgdiplus` will also need to be installed. Because [System.Drawing.Common is no longer supported on Linux](#), this only works on .NET 6 and requires setting the `System.Drawing.EnableUnixSupport` runtime configuration switch.

You can install a recent version of `libgdiplus` by [adding the Mono repository to your system](#).

Next steps

- [How to enable TAB completion for the .NET CLI](#)
- [Tutorial: Create a console application with .NET SDK using Visual Studio Code](#)

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

more information, see [our contributor guide](#).

Install the .NET SDK or the .NET Runtime on Fedora

Article • 11/15/2023

.NET is supported on Fedora and this article describes how to install .NET on Fedora. When a Fedora version falls out of support, .NET is no longer supported with that version.

Install the SDK (which includes the runtime) if you want to develop .NET apps. Or, if you only need to run apps, install the Runtime. If you're installing the Runtime, we suggest you install the **ASP.NET Core Runtime** as it includes both .NET and ASP.NET Core runtimes.

Use the `dotnet --list-sdks` and `dotnet --list-runtimes` commands to see which versions are installed. For more information, see [How to check that .NET is already installed](#).

For more information on installing .NET without a package manager, see one of the following articles:

- [Install the .NET SDK or the .NET Runtime with a script](#).
- [Install the .NET SDK or the .NET Runtime manually](#).

Supported distributions

The following table is a list of currently supported .NET releases and the versions of Fedora they're supported on. These versions remain supported until either the version of [.NET reaches end-of-support](#) or the version of [Fedora reaches end-of-life](#).

[+] Expand table

| Fedora | .NET |
|--------|---------|
| 39 | 8, 7, 6 |
| 38 | 8, 7, 6 |
| 37 | 8, 7, 6 |

The following versions of .NET are  no longer supported:

- .NET 5

- .NET Core 3.1
- .NET Core 3.0
- .NET Core 2.2
- .NET Core 2.1
- .NET Core 2.0

Install .NET 8

Important

.NET 8 was released on November 14, 2023. It may take time for the packages to appear in the package manager feeds.

Install the SDK

The .NET SDK allows you to develop apps with .NET. If you install the .NET SDK, you don't need to install the corresponding runtime. To install the .NET SDK, run the following command:

Bash

```
sudo dnf install dotnet-sdk-8.0
```

Install the runtime

The ASP.NET Core Runtime allows you to run apps that were made with .NET that didn't provide the runtime. The following command installs the ASP.NET Core Runtime, which is the most compatible runtime for .NET. In your terminal, run the following command:

Bash

```
sudo dnf install aspnetcore-runtime-8.0
```

As an alternative to the ASP.NET Core Runtime, you can install the .NET Runtime, which doesn't include ASP.NET Core support: replace `aspnetcore-runtime-8.0` in the previous command with `dotnet-runtime-8.0`:

Bash

```
sudo dnf install dotnet-runtime-8.0
```

Install .NET 7

Install the SDK

The .NET SDK allows you to develop apps with .NET. If you install the .NET SDK, you don't need to install the corresponding runtime. To install the .NET SDK, run the following command:

Bash

```
sudo dnf install dotnet-sdk-7.0
```

Install the runtime

The ASP.NET Core Runtime allows you to run apps that were made with .NET that didn't provide the runtime. The following command installs the ASP.NET Core Runtime, which is the most compatible runtime for .NET. In your terminal, run the following command:

Bash

```
sudo dnf install aspnetcore-runtime-7.0
```

As an alternative to the ASP.NET Core Runtime, you can install the .NET Runtime, which doesn't include ASP.NET Core support: replace `aspnetcore-runtime-7.0` in the previous command with `dotnet-runtime-7.0`:

Bash

```
sudo dnf install dotnet-runtime-7.0
```

Install .NET 6

Install the SDK

The .NET SDK allows you to develop apps with .NET. If you install the .NET SDK, you don't need to install the corresponding runtime. To install the .NET SDK, run the

following command:

```
Bash
```

```
sudo dnf install dotnet-sdk-6.0
```

Install the runtime

The ASP.NET Core Runtime allows you to run apps that were made with .NET that didn't provide the runtime. The following command installs the ASP.NET Core Runtime, which is the most compatible runtime for .NET. In your terminal, run the following command:

```
Bash
```

```
sudo dnf install aspnetcore-runtime-6.0
```

As an alternative to the ASP.NET Core Runtime, you can install the .NET Runtime, which doesn't include ASP.NET Core support: replace `aspnetcore-runtime-6.0` in the previous command with `dotnet-runtime-6.0`:

```
Bash
```

```
sudo dnf install dotnet-runtime-6.0
```

Install preview versions

Preview and release candidate versions of .NET aren't available in package repositories. You can install previews and release candidates of .NET in one of the following ways:

- Scripted install with [*install-dotnet.sh*](#)
- Manual binary extraction

Remove preview versions

When using a package manager to manage your installation of .NET, you may run into a conflict if you've previously installed a preview release. The package manager may interpret the non-preview release as an earlier version of .NET. To install the non-preview release, first uninstall the preview versions. For more information about uninstalling .NET, see [How to remove the .NET Runtime and SDK](#).

Dependencies

When you install with a package manager, these libraries are installed for you. But, if you manually install .NET or you publish a self-contained app, you'll need to make sure these libraries are installed:

- krb5-libs
- libicu
- openssl-libs
- zlib

If the target runtime environment's OpenSSL version is 1.1 or newer, you'll need to install `compat-openssl110`.

Dependencies can be installed with the `yum install` command. The following snippet demonstrates installing the `libicu` library:

Bash

```
sudo yum install libicu
```

For more information about the dependencies, see [Self-contained Linux apps ↗](#).

If the .NET app uses the `System.Drawing.Common` assembly, `libgdiplus` will also need to be installed. Because [System.Drawing.Common is no longer supported on Linux](#), this only works on .NET 6 and requires setting the `System.Drawing.EnableUnixSupport` runtime configuration switch.

You can install a recent version of `libgdiplus` by [adding the Mono repository to your system ↗](#).

Install on older distributions

Older versions of Fedora don't contain .NET Core in the default package repositories. You can install .NET with the [`dotnet-install.sh` script](#), or use Microsoft's repository to install .NET:

1. First, add the Microsoft signing key to your list of trusted keys.

Bash

```
sudo rpm --import https://packages.microsoft.com/keys/microsoft.asc
```

2. Next, add the Microsoft package repository. The source of the repository is based on your version of Fedora.

[+] Expand table

| Fedora Version | Package repository |
|----------------|---|
| 36 | https://packages.microsoft.com/config/fedora/36/prod.repo |
| 35 | https://packages.microsoft.com/config/fedora/35/prod.repo |
| 34 | https://packages.microsoft.com/config/fedora/34/prod.repo |
| 33 | https://packages.microsoft.com/config/fedora/33/prod.repo |
| 32 | https://packages.microsoft.com/config/fedora/32/prod.repo |
| 31 | https://packages.microsoft.com/config/fedora/31/prod.repo |
| 30 | https://packages.microsoft.com/config/fedora/30/prod.repo |
| 29 | https://packages.microsoft.com/config/fedora/29/prod.repo |
| 28 | https://packages.microsoft.com/config/fedora/28/prod.repo |
| 27 | https://packages.microsoft.com/config/fedora/27/prod.repo |

Bash

```
sudo wget -O /etc/yum.repos.d/microsoft-prod.repo  
https://packages.microsoft.com/config/fedora/31/prod.repo
```

Install the SDK

The .NET SDK allows you to develop apps with .NET. If you install the .NET SDK, you don't need to install the corresponding runtime. To install the .NET SDK, run the following command:

Bash

```
sudo dnf install dotnet-sdk-7.0
```

Install the runtime

The ASP.NET Core Runtime allows you to run apps that were made with .NET that didn't provide the runtime. The following command installs the ASP.NET Core Runtime, which is the most compatible runtime for .NET. In your terminal, run the following command:

```
Bash
```

```
sudo dnf install aspnetcore-runtime-7.0
```

As an alternative to the ASP.NET Core Runtime, you can install the .NET Runtime, which doesn't include ASP.NET Core support: replace `aspnetcore-runtime-7.0` in the previous command with `dotnet-runtime-7.0`:

```
Bash
```

```
sudo dnf install dotnet-runtime-7.0
```

How to install other versions

All versions of .NET are available for download at

<https://dotnet.microsoft.com/download/dotnet>, but require [manual installation](#). You can try and use the package manager to install a different version of .NET. However, the requested version may not be available.

The packages added to package manager feeds are named in a hackable format, for example: `{product}-{type}-{version}`.

- **product**

The type of .NET product to install. Valid options are:

- `dotnet`
- `aspnetcore`

- **type**

Chooses the SDK or the runtime. Valid options are:

- `sdk` (only available for the `dotnet` product)
- `runtime`

- **version**

The version of the SDK or runtime to install. This article will always give the instructions for the latest supported version. Valid options are any released version, such as:

- `8.0`
- `6.0`

- 3.1
- 2.1

It's possible the SDK/runtime you're trying to download is not available for your Linux distribution. For a list of supported distributions, see [Install .NET on Linux](#).

Examples

- Install the ASP.NET Core 8.0 runtime: `aspnetcore-runtime-8.0`
- Install the .NET Core 2.1 runtime: `dotnet-runtime-2.1`
- Install the .NET 5 SDK: `dotnet-sdk-5.0`
- Install the .NET Core 3.1 SDK: `dotnet-sdk-3.1`

Package missing

If the package-version combination doesn't work, it's not available. For example, there isn't an ASP.NET Core SDK, the SDK components are included with the .NET SDK. The value `aspnetcore-sdk-8.0` is incorrect and should be `dotnet-sdk-8.0`. For a list of Linux distributions supported by .NET, see [.NET dependencies and requirements](#).

Troubleshoot the package manager

This section provides information on common errors you may get while using the package manager to install .NET or .NET Core.

Unable to find package

For more information on installing .NET without a package manager, see one of the following articles:

- [Install the .NET SDK or the .NET Runtime with a script](#).
- [Install the .NET SDK or the .NET Runtime manually](#).

Failed to fetch

While installing the .NET package, you may see an error similar to `signature verification failed for file 'repomd.xml' from repository 'packages-microsoft-com-prod'`. Generally speaking, this error means that the package feed for .NET is being upgraded with newer package versions, and that you should try again later. During an upgrade, the package feed should not be unavailable for more than 2 hours. If you

continually receive this error for more than 2 hours, please file an issue at <https://github.com/dotnet/core/issues>.

Errors related to missing `fxr`, `libhostfxr.so`, `FrameworkList.xml`, or `/usr/share/dotnet`

For more information about solving these problems, see [Troubleshoot `fxr`, `libhostfxr.so`, and `FrameworkList.xml` errors](#).

Next steps

- [How to enable TAB completion for the .NET CLI](#)
- [Tutorial: Create a console application with .NET SDK using Visual Studio Code](#)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

.NET is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

Install the .NET SDK or the .NET Runtime on openSUSE

Article • 11/14/2023

.NET is supported on openSUSE. This article describes how to install .NET on openSUSE.

Install the SDK (which includes the runtime) if you want to develop .NET apps. Or, if you only need to run apps, install the Runtime. If you're installing the Runtime, we suggest you install the **ASP.NET Core Runtime** as it includes both .NET and ASP.NET Core runtimes.

Use the `dotnet --list-sdks` and `dotnet --list-runtimes` commands to see which versions are installed. For more information, see [How to check that .NET is already installed](#).

ⓘ Important

Package manager installs are only supported on the **x64** architecture. Other architectures, such as **Arm**, must install .NET by some other means such as with Snap, an installer script, or through a manual binary installation.

For more information on installing .NET **without a package manager**, see one of the following articles:

- [Alternatively install .NET with install-dotnet script](#).
- [Manually install .NET](#)

Supported distributions

The following table is a list of currently supported .NET releases on openSUSE 15. These versions remain supported until either the version of [.NET reaches end-of-support](#) or the version of openSUSE is no longer supported.

| openSUSE | .NET |
|----------|---------|
| 15.4+ | 8, 7, 6 |

The following versions of .NET are  no longer supported:

- .NET 5

- .NET Core 3.1
- .NET Core 3.0
- .NET Core 2.2
- .NET Core 2.1
- .NET Core 2.0

Install preview versions

Preview and release candidate versions of .NET aren't available in package repositories. You can install previews and release candidates of .NET in one of the following ways:

- Scripted install with [*install-dotnet.sh*](#)
- Manual binary extraction

Remove preview versions

When using a package manager to manage your installation of .NET, you may run into a conflict if you've previously installed a preview release. The package manager may interpret the non-preview release as an earlier version of .NET. To install the non-preview release, first uninstall the preview versions. For more information about uninstalling .NET, see [How to remove the .NET Runtime and SDK](#).

openSUSE 15

Before you install .NET, run the following commands to add the Microsoft package signing key to your list of trusted keys and add the Microsoft package repository. Open a terminal and run the following commands:

Bash

```
sudo zypper install libicu
sudo rpm --import https://packages.microsoft.com/keys/microsoft.asc
wget https://packages.microsoft.com/config/opensuse/15/prod.repo
sudo mv prod.repo /etc/zypp/repos.d/microsoft-prod.repo
sudo chown root:root /etc/zypp/repos.d/microsoft-prod.repo
```

Install the SDK

The .NET SDK allows you to develop apps with .NET. If you install the .NET SDK, you don't need to install the corresponding runtime. To install the .NET SDK, run the following command:

```
Bash
```

```
sudo zypper install dotnet-sdk-8.0
```

Install the runtime

The ASP.NET Core Runtime allows you to run apps that were made with .NET that didn't provide the runtime. The following command installs the ASP.NET Core Runtime, which is the most compatible runtime for .NET. In your terminal, run the following command:

```
Bash
```

```
sudo zypper install aspnetcore-runtime-8.0
```

As an alternative to the ASP.NET Core Runtime, you can install the .NET Runtime, which doesn't include ASP.NET Core support: replace `aspnetcore-runtime-8.0` in the previous command with `dotnet-runtime-8.0`:

```
Bash
```

```
sudo zypper install dotnet-runtime-8.0
```

How to install other versions

All versions of .NET are available for download at

<https://dotnet.microsoft.com/download/dotnet>, but require [manual installation](#). You can try and use the package manager to install a different version of .NET. However, the requested version may not be available.

The packages added to package manager feeds are named in a hackable format, for example: `{product}-{type}-{version}`.

- **product**

The type of .NET product to install. Valid options are:

- `dotnet`
- `aspnetcore`

- **type**

Chooses the SDK or the runtime. Valid options are:

- `sdk` (only available for the `dotnet` product)
- `runtime`

- **version**

The version of the SDK or runtime to install. This article will always give the instructions for the latest supported version. Valid options are any released version, such as:

- 8.0
- 6.0
- 3.1
- 2.1

It's possible the SDK/runtime you're trying to download is not available for your Linux distribution. For a list of supported distributions, see [Install .NET on Linux](#).

Examples

- Install the ASP.NET Core 8.0 runtime: `aspnetcore-runtime-8.0`
- Install the .NET Core 2.1 runtime: `dotnet-runtime-2.1`
- Install the .NET 5 SDK: `dotnet-sdk-5.0`
- Install the .NET Core 3.1 SDK: `dotnet-sdk-3.1`

Package missing

If the package-version combination doesn't work, it's not available. For example, there isn't an ASP.NET Core SDK, the SDK components are included with the .NET SDK. The value `aspnetcore-sdk-8.0` is incorrect and should be `dotnet-sdk-8.0`. For a list of Linux distributions supported by .NET, see [.NET dependencies and requirements](#).

Troubleshoot the package manager

This section provides information on common errors you may get while using the package manager to install .NET.

Unable to find package

 **Important**

Package manager installs are only supported on the **x64** architecture. Other architectures, such as **Arm**, must install .NET by some other means such as with Snap, an installer script, or through a manual binary installation.

For more information on installing .NET **without a package manager**, see one of the following articles:

- [Alternatively install .NET with install-dotnet script.](#)
- [Manually install .NET](#)

Failed to fetch

While installing the .NET package, you may see an error similar to `signature verification failed for file 'repomd.xml' from repository 'packages-microsoft-com-prod'`. Generally speaking, this error means that the package feed for .NET is being upgraded with newer package versions, and that you should try again later. During an upgrade, the package feed should not be unavailable for more than 2 hours. If you continually receive this error for more than 2 hours, please file an issue at <https://github.com/dotnet/core/issues>.

Dependencies

When you install with a package manager, these libraries are installed for you. But, if you manually install .NET or you publish a self-contained app, you'll need to make sure these libraries are installed:

- `krb5`
- `libicu`
- `libopenssl1_0_0`

If the target runtime environment's OpenSSL version is 1.1 or newer, you'll need to install `compat-openssl110`.

Dependencies can be installed with the `zypper install` command. The following snippet demonstrates installing the `krb5` library:

```
Bash
```

```
sudo zypper install krb5
```

For more information about the dependencies, see [Self-contained Linux apps](#).

If the .NET app uses the `System.Drawing.Common` assembly, `libgdiplus` will also need to be installed. Because [System.Drawing.Common is no longer supported on Linux](#), this only works on .NET 6 and requires setting the `System.Drawing.EnableUnixSupport` runtime configuration switch.

Next steps

- [How to enable TAB completion for the .NET CLI](#)
- [Tutorial: Create a console application with .NET SDK using Visual Studio Code](#)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

Install the .NET SDK or the .NET Runtime on RHEL and CentOS Stream

Article • 11/15/2023

ⓘ Important

.NET 8 was released on November 14, 2023. It may take time for the packages to appear in the package manager feeds.

.NET is supported on Red Hat Enterprise Linux (RHEL). This article describes how to install .NET on RHEL and CentOS Stream.

Install the SDK (which includes the runtime) if you want to develop .NET apps. Or, if you only need to run apps, install the Runtime. If you're installing the Runtime, we suggest you install the **ASP.NET Core Runtime** as it includes both .NET and ASP.NET Core runtimes.

Use the `dotnet --list-sdks` and `dotnet --list-runtimes` commands to see which versions are installed. For more information, see [How to check that .NET is already installed](#).

Register your Red Hat subscription

To install .NET from Red Hat on RHEL, you first need to register using the Red Hat Subscription Manager. If this hasn't been done on your system, or if you're unsure, see the [Red Hat Product Documentation for .NET](#).

ⓘ Important

This doesn't apply to CentOS Stream.

Supported distributions

The following table is a list of currently supported .NET releases on both RHEL and CentOS Stream. These versions remain supported until either the version of [.NET reaches end-of-support](#) or the Linux distribution is no longer supported.

| Distribution | .NET |
|-----------------|---------|
| RHEL 9 (9.1) | 8, 7, 6 |
| RHEL 8 (8.7) | 8, 7, 6 |
| RHEL 7 | 6 |
| CentOS Stream 9 | 8, 7, 6 |
| CentOS Stream 8 | 8, 7, 6 |

The following versions of .NET are  no longer supported:

- .NET 5
- .NET Core 3.1
- .NET Core 3.0
- .NET Core 2.2
- .NET Core 2.1
- .NET Core 2.0

Install preview versions

Preview and release candidate versions of .NET aren't available in package repositories. You can install previews and release candidates of .NET in one of the following ways:

- Scripted install with [install-dotnet.sh](#)
- Manual binary extraction

Remove preview versions

When using a package manager to manage your installation of .NET, you may run into a conflict if you've previously installed a preview release. The package manager may interpret the non-preview release as an earlier version of .NET. To install the non-preview release, first uninstall the preview versions. For more information about uninstalling .NET, see [How to remove the .NET Runtime and SDK](#).

RHEL 9

.NET is included in the AppStream repositories for RHEL 9.

Important

.NET 8 was released on November 14, 2023. It may take time for the packages to appear in the package manager feeds.

Install the SDK

The .NET SDK allows you to develop apps with .NET. If you install the .NET SDK, you don't need to install the corresponding runtime. To install the .NET SDK, run the following command:

Bash

```
sudo dnf install dotnet-sdk-8.0
```

Install the runtime

The ASP.NET Core Runtime allows you to run apps that were made with .NET that didn't provide the runtime. The following command installs the ASP.NET Core Runtime, which is the most compatible runtime for .NET. In your terminal, run the following command:

Bash

```
sudo dnf install aspnetcore-runtime-8.0
```

As an alternative to the ASP.NET Core Runtime, you can install the .NET Runtime, which doesn't include ASP.NET Core support: replace `aspnetcore-runtime-8.0` in the previous command with `dotnet-runtime-8.0`:

Bash

```
sudo dnf install dotnet-runtime-8.0
```

RHEL 8

.NET is included in the AppStream repositories for RHEL 8.

Important

.NET 8 was released on November 14, 2023. It may take time for the packages to appear in the package manager feeds.

Install the SDK

The .NET SDK allows you to develop apps with .NET. If you install the .NET SDK, you don't need to install the corresponding runtime. To install the .NET SDK, run the following command:

Bash

```
sudo dnf install dotnet-sdk-8.0
```

Install the runtime

The ASP.NET Core Runtime allows you to run apps that were made with .NET that didn't provide the runtime. The following command installs the ASP.NET Core Runtime, which is the most compatible runtime for .NET. In your terminal, run the following command:

Bash

```
sudo dnf install aspnetcore-runtime-8.0
```

As an alternative to the ASP.NET Core Runtime, you can install the .NET Runtime, which doesn't include ASP.NET Core support: replace `aspnetcore-runtime-8.0` in the previous command with `dotnet-runtime-8.0`:

Bash

```
sudo dnf install dotnet-runtime-8.0
```

RHEL 7 ✗ .NET 8

.NET 8 isn't compatible with RHEL 7 and doesn't work.

RHEL 7 ✗ .NET 7

.NET 7 isn't officially supported on RHEL 7. To install .NET 7, see [Install .NET on Linux by using an install script or by extracting binaries](#).

RHEL 7 ✓ .NET 6

The following command installs the `scl-utils` package:

Bash

```
sudo yum install scl-utils
```

Install the SDK

The .NET SDK allows you to develop apps with .NET. If you install the .NET SDK, you don't need to install the corresponding runtime. To install .NET SDK, run the following commands:

Bash

```
subscription-manager repos --enable=rhel-7-server-dotnet-rpms
yum install rh-dotnet60 -y
scl enable rh-dotnet60 bash
```

Red Hat does not recommend permanently enabling `rh-dotnet60` because it may affect other programs. If you want to enable `rh-dotnet` permanently, add the following line to your `~/.bashrc` file.

Bash

```
source scl_source enable rh-dotnet60
```

Install the runtime

The .NET Runtime allows you to run apps that were made with .NET that didn't include the runtime. The commands below install the ASP.NET Core Runtime, which is the most compatible runtime for .NET Core. In your terminal, run the following commands.

Bash

```
subscription-manager repos --enable=rhel-7-server-dotnet-rpms
yum install rh-dotnet60-aspnetcore-runtime-6.0 -y
scl enable rh-dotnet60 bash
```

Red Hat does not recommend permanently enabling `rh-dotnet60` because it may affect other programs. If you want to enable `rh-dotnet60` permanently, add the following line

to your `~/.bashrc` file.

Bash

```
source scl_source enable rh-dotnet60
```

As an alternative to the ASP.NET Core Runtime, you can install the .NET Runtime that doesn't include ASP.NET Core support: replace `rh-dotnet60-aspnetcore-runtime-6.0` in the preceding command with `rh-dotnet60-dotnet-runtime-6.0`.

CentOS Stream 9 ✓

.NET is included in the AppStream repositories for CentOS Stream 9.

Install the SDK

The .NET SDK allows you to develop apps with .NET. If you install the .NET SDK, you don't need to install the corresponding runtime. To install the .NET SDK, run the following command:

Bash

```
sudo dnf install dotnet-sdk-8.0
```

Install the runtime

The ASP.NET Core Runtime allows you to run apps that were made with .NET that didn't provide the runtime. The following command installs the ASP.NET Core Runtime, which is the most compatible runtime for .NET. In your terminal, run the following command:

Bash

```
sudo dnf install aspnetcore-runtime-8.0
```

As an alternative to the ASP.NET Core Runtime, you can install the .NET Runtime, which doesn't include ASP.NET Core support: replace `aspnetcore-runtime-8.0` in the previous command with `dotnet-runtime-8.0`:

Bash

```
sudo dnf install dotnet-runtime-8.0
```

CentOS Stream 8 ✓

Use the Microsoft repository to install .NET:

Bash

```
sudo rpm -Uvh https://packages.microsoft.com/config/centos/8/packages-microsoft-prod.rpm
sudo yum install dotnet-sdk-8.0
```

Dependencies

When you install with a package manager, these libraries are installed for you. But, if you manually install .NET or you publish a self-contained app, you'll need to make sure these libraries are installed:

- krb5-libs
- libicu
- openssl-libs
- zlib

If the target runtime environment's OpenSSL version is 1.1 or newer, you'll need to install `compat-openssl110`.

Dependencies can be installed with the `yum install` command. The following snippet demonstrates installing the `libicu` library:

Bash

```
sudo yum install libicu
```

For more information about the dependencies, see [Self-contained Linux apps ↗](#).

If the .NET app uses the `System.Drawing.Common` assembly, `libgdiplus` will also need to be installed. Because [System.Drawing.Common is no longer supported on Linux](#), this only works on .NET 6 and requires setting the `System.Drawing.EnableUnixSupport` runtime configuration switch.

You can install a recent version of *libgdiplus* by [adding the Mono repository to your system](#).

How to install other versions

Consult the [Red Hat documentation for .NET](#) on the steps required to install other releases of .NET.

Troubleshoot the package manager

This section provides information on common errors you may get while using the package manager to install .NET or .NET Core.

Errors related to missing `fxr`, `libhostfxr.so`, or `FrameworkList.xml`

For more information about solving these problems, see [Troubleshoot fxr, libhostfxr.so, and FrameworkList.xml errors](#).

Next steps

- [How to enable TAB completion for the .NET CLI](#)
- [Tutorial: Create a console application with .NET SDK using Visual Studio Code](#)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



.NET feedback

.NET is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

Install the .NET SDK or the .NET Runtime on SLES

Article • 11/14/2023

.NET is supported on SLES. This article describes how to install .NET on SLES.

Install the SDK (which includes the runtime) if you want to develop .NET apps. Or, if you only need to run apps, install the Runtime. If you're installing the Runtime, we suggest you install the **ASP.NET Core Runtime** as it includes both .NET and ASP.NET Core runtimes.

Use the `dotnet --list-sdks` and `dotnet --list-runtimes` commands to see which versions are installed. For more information, see [How to check that .NET is already installed](#).

Supported distributions

The following table is a list of currently supported .NET releases on both SLES 12 SP2 and SLES 15. These versions remain supported until either the version of [.NET reaches end-of-support](#) or the version of SLES is no longer supported.

| SLES | .NET |
|--------|---------|
| 15 | 8, 7, 6 |
| 12 SP5 | 8, 7, 6 |

The following versions of .NET are  no longer supported:

- .NET 5
- .NET Core 3.1
- .NET Core 3.0
- .NET Core 2.2
- .NET Core 2.1
- .NET Core 2.0

Install preview versions

Preview and release candidate versions of .NET aren't available in package repositories. You can install previews and release candidates of .NET in one of the following ways:

- Scripted install with *install-dotnet.sh*
- Manual binary extraction

Remove preview versions

When using a package manager to manage your installation of .NET, you may run into a conflict if you've previously installed a preview release. The package manager may interpret the non-preview release as an earlier version of .NET. To install the non-preview release, first uninstall the preview versions. For more information about uninstalling .NET, see [How to remove the .NET Runtime and SDK](#).

SLES 15

Before you install .NET, run the following commands to add the Microsoft package signing key to your list of trusted keys and add the Microsoft package repository. Open a terminal and run the following commands:

Bash

```
sudo rpm -Uvh https://packages.microsoft.com/config/sles/15/packages-microsoft-prod.rpm
```

Currently, the SLES 15 Microsoft repository setup package installs the *microsoft-prod.repo* file to the wrong directory, preventing zypper from finding the .NET packages. To fix this problem, create a symlink in the correct directory.

Bash

```
sudo ln -s /etc/yum.repos.d/microsoft-prod.repo /etc/zypp/repos.d/microsoft-prod.repo
```

Install the SDK

The .NET SDK allows you to develop apps with .NET. If you install the .NET SDK, you don't need to install the corresponding runtime. To install the .NET SDK, run the following command:

Bash

```
sudo zypper install dotnet-sdk-8.0
```

Install the runtime

The ASP.NET Core Runtime allows you to run apps that were made with .NET that didn't provide the runtime. The following command installs the ASP.NET Core Runtime, which is the most compatible runtime for .NET. In your terminal, run the following command:

Bash

```
sudo zypper install aspnetcore-runtime-8.0
```

As an alternative to the ASP.NET Core Runtime, you can install the .NET Runtime, which doesn't include ASP.NET Core support: replace `aspnetcore-runtime-8.0` in the previous command with `dotnet-runtime-8.0`:

Bash

```
sudo zypper install dotnet-runtime-8.0
```

SLES 12

.NET requires SP2 as a minimum for the SLES 12 family.

Before you install .NET, run the following commands to add the Microsoft package signing key to your list of trusted keys and add the Microsoft package repository. Open a terminal and run the following commands:

Bash

```
sudo rpm -Uvh https://packages.microsoft.com/config/sles/12/packages-microsoft-prod.rpm
```

Install the SDK

The .NET SDK allows you to develop apps with .NET. If you install the .NET SDK, you don't need to install the corresponding runtime. To install the .NET SDK, run the following command:

Bash

```
sudo zypper install dotnet-sdk-8.0
```

Install the runtime

The ASP.NET Core Runtime allows you to run apps that were made with .NET that didn't provide the runtime. The following command installs the ASP.NET Core Runtime, which is the most compatible runtime for .NET. In your terminal, run the following command:

Bash

```
sudo zypper install aspnetcore-runtime-8.0
```

As an alternative to the ASP.NET Core Runtime, you can install the .NET Runtime, which doesn't include ASP.NET Core support: replace `aspnetcore-runtime-8.0` in the previous command with `dotnet-runtime-8.0`:

Bash

```
sudo zypper install dotnet-runtime-8.0
```

How to install other versions

All versions of .NET are available for download at

<https://dotnet.microsoft.com/download/dotnet>, but require [manual installation](#). You can try and use the package manager to install a different version of .NET. However, the requested version may not be available.

The packages added to package manager feeds are named in a hackable format, for example: `{product}-{type}-{version}`.

- **product**

The type of .NET product to install. Valid options are:

- `dotnet`
- `aspnetcore`

- **type**

Chooses the SDK or the runtime. Valid options are:

- `sdk` (only available for the `dotnet` product)
- `runtime`

- **version**

The version of the SDK or runtime to install. This article will always give the instructions for the latest supported version. Valid options are any released version, such as:

- 8.0
- 6.0
- 3.1
- 2.1

It's possible the SDK/runtime you're trying to download is not available for your Linux distribution. For a list of supported distributions, see [Install .NET on Linux](#).

Examples

- Install the ASP.NET Core 8.0 runtime: `aspnetcore-runtime-8.0`
- Install the .NET Core 2.1 runtime: `dotnet-runtime-2.1`
- Install the .NET 5 SDK: `dotnet-sdk-5.0`
- Install the .NET Core 3.1 SDK: `dotnet-sdk-3.1`

Package missing

If the package-version combination doesn't work, it's not available. For example, there isn't an ASP.NET Core SDK, the SDK components are included with the .NET SDK. The value `aspnetcore-sdk-8.0` is incorrect and should be `dotnet-sdk-8.0`. For a list of Linux distributions supported by .NET, see [.NET dependencies and requirements](#).

Troubleshoot the package manager

This section provides information on common errors you may get while using the package manager to install .NET.

Failed to fetch

While installing the .NET package, you may see an error similar to `signature verification failed for file 'repomd.xml' from repository 'packages-microsoft-com-prod'`. Generally speaking, this error means that the package feed for .NET is being upgraded with newer package versions, and that you should try again later. During an upgrade, the package feed should not be unavailable for more than 2 hours. If you continually receive this error for more than 2 hours, please file an issue at <https://github.com/dotnet/core/issues>.

Dependencies

When you install with a package manager, these libraries are installed for you. But, if you manually install .NET or you publish a self-contained app, you'll need to make sure these libraries are installed:

- krb5
- libicu
- libopenssl1_1

If the target runtime environment's OpenSSL version is 1.1 or newer, you'll need to install `compat-openssl10`.

Dependencies can be installed with the `zypper install` command. The following snippet demonstrates installing the `krb5` library:

```
Bash
```

```
sudo zypper install krb5
```

For more information about the dependencies, see [Self-contained Linux apps ↗](#).

If the .NET app uses the `System.Drawing.Common` assembly, `libgdiplus` will also need to be installed. Because [System.Drawing.Common is no longer supported on Linux](#), this only works on .NET 6 and requires setting the `System.Drawing.EnableUnixSupport` runtime configuration switch.

Next steps

- [How to enable TAB completion for the .NET CLI](#)
- [Tutorial: Create a console application with .NET SDK using Visual Studio Code](#)



Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



.NET feedback

The .NET documentation is open source. Provide feedback here.



[Open a documentation issue](#)



[Provide product feedback](#)

Install .NET on Linux by using an install script or by extracting binaries

Article • 12/30/2023

This article demonstrates how to install the .NET SDK or the .NET Runtime on Linux by using the install script or by extracting the binaries. For a list of distributions that support the built-in package manager, see [Install .NET on Linux](#).

Install the SDK (which includes the runtime) if you want to develop .NET apps. Or, if you only need to run apps, install the Runtime. If you're installing the Runtime, we suggest you install the **ASP.NET Core Runtime** as it includes both .NET and ASP.NET Core runtimes.

Use the `dotnet --list-sdks` and `dotnet --list-runtimes` commands to see which versions are installed. For more information, see [How to check that .NET is already installed](#).

.NET releases

There are two types of supported releases, Long Term Support (LTS) releases or Standard Term Support (STS). The quality of all releases is the same. The only difference is the length of support. LTS releases get free support and patches for 3 years. STS releases get free support and patches for 18 months. For more information, see [.NET Support Policy](#).

The following table lists the support status of each version of .NET (and .NET Core):

[] Expand table

| ✓ Supported | ✗ Unsupported |
|-------------|---------------|
| 8 (LTS) | 5 |
| 7 (STS) | 3.1 |
| 6 (LTS) | 3.0 |
| | 2.2 |
| | 2.1 |
| | 2.0 |

| ✓ Supported | ✗ Unsupported |
|-------------|---------------|
| | 1.1 |
| | 1.0 |

Dependencies

It's possible that when you install .NET, specific dependencies may not be installed, such as when [manually installing](#). The following list details Linux distributions that are supported by Microsoft and have dependencies you may need to install. Check the distribution page for more information:

- [Alpine](#)
- [Debian](#)
- [CentOS](#)
- [Fedora](#)
- [RHEL and CentOS Stream](#)
- [SLES](#)
- [Ubuntu](#)

For generic information about the dependencies, see [Self-contained Linux apps ↗](#).

RPM dependencies

If your distribution wasn't previously listed, and is RPM-based, you may need the following dependencies:

- krb5-libs
- libicu
- openssl-libs

If the target runtime environment's OpenSSL version is 1.1 or newer, install `compat-openssl110`.

DEB dependencies

If your distribution wasn't previously listed, and is debian-based, you may need the following dependencies:

- libc6
- libgcc1

- libgssapi-krb5-2
- libicu67
- libssl1.1
- libstdc++6
- zlib1g

Common dependencies

If the .NET app uses the *System.Drawing.Common* assembly, libgdiplus will also need to be installed. Because [System.Drawing.Common is no longer supported on Linux](#), this only works on .NET 6 and requires setting the `System.Drawing.EnableUnixSupport` runtime configuration switch.

You can usually install a recent version of *libgdiplus* by [adding the Mono repository to your system ↗](#).

Scripted install

The [dotnet-install scripts](#) are used for automation and non-admin installs of the **SDK** and **Runtime**. You can download the script from <https://dot.net/v1/dotnet-install.sh> ↗. When .NET is installed in this way, you must install the dependencies required by your Linux distribution. Use the links in the [Install .NET on Linux](#) article for your specific Linux distribution.

ⓘ Important

Bash is required to run the script.

You can download the script with `wget`:

Bash

```
wget https://dot.net/v1/dotnet-install.sh -O dotnet-install.sh
```

Before running this script, make sure you grant permission for this script to run as an executable:

Bash

```
chmod +x ./dotnet-install.sh
```

The script defaults to installing the latest [long term support \(LTS\)](#) SDK version, which is .NET 8. To install the latest release, which might not be an (LTS) version, use the `--version latest` parameter.

Bash

```
./dotnet-install.sh --version latest
```

To install .NET Runtime instead of the SDK, use the `--runtime` parameter.

Bash

```
./dotnet-install.sh --version latest --runtime aspnetcore
```

You can install a specific major version with the `--channel` parameter to indicate the specific version. The following command installs .NET 8.0 SDK.

Bash

```
./dotnet-install.sh --channel 8.0
```

For more information, see [dotnet-install scripts reference](#).

To enable .NET on the command line, see [Set environment variables system-wide](#).

Manual install

As an alternative to the package managers, you can download and manually install the SDK and runtime. Manual installation is commonly used as part of continuous integration testing or on an unsupported Linux distribution. For a developer or user, it's better to use a package manager.

Download a **binary** release for either the SDK or the runtime from one of the following sites. The .NET SDK includes the corresponding runtime:

- [.NET 8 downloads](#)
- [.NET 7 downloads](#)
- [.NET 6 downloads](#)
- [All .NET Core downloads](#)

Extract the downloaded file and use the `export` command to set `DOTNET_ROOT` to the extracted folder's location and then ensure .NET is in PATH. Exporting `DOTNET_ROOT`

makes the .NET CLI commands available in the terminal. For more information about .NET environment variables, see [.NET SDK and CLI environment variables](#).

Different versions of .NET can be extracted to the same folder, which coexist side-by-side.

Example

The following commands use Bash to set the environment variable `DOTNET_ROOT` to the current working directory followed by `.dotnet`. That directory is created if it doesn't exist. The `DOTNET_FILE` environment variable is the filename of the .NET binary release you want to install. This file is extracted to the `DOTNET_ROOT` directory. Both the `DOTNET_ROOT` directory and its `tools` subdirectory are added to the `PATH` environment variable.

ⓘ Important

If you run these commands, remember to change the `DOTNET_FILE` value to the name of the .NET binary you downloaded.

Bash

```
DOTNET_FILE=dotnet-sdk-8.0.100-linux-x64.tar.gz
export DOTNET_ROOT=$(pwd)/.dotnet

mkdir -p "$DOTNET_ROOT" && tar zxf "$DOTNET_FILE" -C "$DOTNET_ROOT"

export PATH=$PATH:$DOTNET_ROOT:$DOTNET_ROOT/tools
```

You can install more than one version of .NET in the same folder.

You can also install .NET to the home directory identified by the `HOME` variable or `~` path:

Bash

```
export DOTNET_ROOT=$HOME/.dotnet
```

Verify downloaded binaries

After downloading an installer, verify it to make sure that the file hasn't been changed or corrupted. You can verify the checksum on your computer and then compare it to

what was reported on the download website.

When you download an installer or binary from an official download page, the checksum for the file is displayed. Select the **Copy** button to copy the checksum value to your clipboard.

The screenshot shows the download page for the .NET 8.0 SDK (v8.0.100) - Windows x64 Installer. At the top, it says "Thanks for downloading". Below that, there's a note about Visual Studio compatibility. A "Direct link" is provided with a "Copy" button next to it. Underneath, a "Checksum (SHA512)" is listed as "248acec95b381e5302255310fb9396267fd74a4a2dc2c3a5989031969cb31f8270cbd14bda1bc0352ac90f8138bddad1a58e4af1e56cc4a1613b1cf2854b5", with a "Copy" button next to it. A red box highlights the "Copy" button for the checksum.

Use the `sha512sum` command to print the checksum of the file you've downloaded. For example, the following command reports the checksum of the *dotnet-sdk-8.0.100-linux-x64.tar.gz* file:

Bash

```
$ sha512sum dotnet-sdk-8.0.100-linux-x64.tar.gz
13905ea20191e70baeba50b0e9bbe5f752a7c34587878ee104744f9fb453bfe439994d389697
22bdae7f60ee047d75dda8636f3ab62659450e9cd4024f38b2a5  dotnet-sdk-8.0.100-
linux-x64.tar.gz
```

Compare the checksum with the value provided by the download site.

ⓘ Important

Even though a Linux file is shown in these examples, this information equally applies to macOS.

Use a checksum file to validate

The .NET release notes contain a link to a checksum file you can use to validate your downloaded file. The following steps describe how to download the checksum file and validate a .NET install binary:

1. The release notes page for .NET 8 on GitHub at <https://github.com/dotnet/core/tree/main/release-notes/8.0> contains a section named **Releases**. The table in that section links to the downloads and checksum files for each .NET 8 release:

| Date | Release |
|------------|---------------------------------|
| 2023/11/14 | 8.0.0 |
| 2023/10/10 | 8.0.0 RC 2 |
| 2023/09/12 | 8.0.0 RC 1 |
| 2023/08/08 | 8.0.0 Preview 7 |
| 2023/07/11 | 8.0.0 Preview 6 |

2. Select the link for the version of .NET that you downloaded. The previous section used .NET SDK 8.0.100, which is in the .NET 8.0.0 release.
3. In the release page, you can see the .NET Runtime and .NET SDK version, and a link to the checksum file:

| .NET 8.0.0 - November 14, 2023 | | | | | | |
|---|---|---|---|--|--|---|
| The .NET 8.0.0 and .NET SDK 8.0.100 releases are available for download. The latest 8.0 release is always listed at .NET 8.0 Releases . | | | | | | |
| Downloads | | | | | | |
| | SDK Installer ¹ | SDK Binaries ¹ | Runtime Installer | Runtime Binaries | ASP.NET Core Runtime | Windows Desktop Runtime |
| Windows | x86 x64 Arm64 | x86 x64 Arm64 | x86 x64 Arm64 | x86 x64 Arm64 | x86 x64 Hosting Bundle² | x86 x64 Arm64 |
| macOS | x64 ARM64 | x64 ARM64 | x64 ARM64 | x64 ARM64 | x64 ARM64 | - |
| Linux | Snap and Package Manager | x64 Arm Arm64 Arm32 Alpine x64 Alpine | Packages (x64) | x64 Arm Arm64 Arm32 Alpine Arm64 Alpine x64 Alpine | x64¹ Arm¹ Arm64¹ x64 Alpine | - |
| | Checksums | Checksums | Checksums | Checksums | Checksums | Checksums |

4. Copy the link to the checksum file.
5. Use the following script, but replace the link to download the appropriate checksum file:

```
Bash
```

```
curl -O https://dotnetcli.blob.core.windows.net/dotnet/checksums/8.0.0-
```

```
sha.txt
```

- With both the checksum file and the .NET release file downloaded to the same directory, use the `sha512sum -c {file} --ignore-missing` command to validate the downloaded file.

When validation passes, you see the file printed with the **OK** status:

```
Bash
```

```
$ sha512sum -c 8.0.0-sha.txt --ignore-missing  
dotnet-sdk-8.0.100-linux-x64.tar.gz: OK
```

If you see the file marked as **FAILED**, the file you downloaded isn't valid and shouldn't be used.

```
Bash
```

```
$ sha512sum -c 8.0.0-sha.txt --ignore-missing  
dotnet-sdk-8.0.100-linux-x64.tar.gz: FAILED  
sha512sum: WARNING: 1 computed checksum did NOT match  
sha512sum: 8.0.0-sha.txt: no file was verified
```

Set environment variables system-wide

If you used the previous install script, the variables set only apply to your current terminal session. Add them to your shell profile. There are many different shells available for Linux and each has a different profile. For example:

- **Bash Shell:** `~/.bash_profile` or `~/.bashrc`
- **Korn Shell:** `~/.kshrc` or `.profile`
- **Z Shell:** `~/.zshrc` or `.zprofile`

Set the following two environment variables in your shell profile:

- `DOTNET_ROOT`

This variable is set to the folder .NET was installed to, such as `$HOME/.dotnet`:

```
Bash
```

```
export DOTNET_ROOT=$HOME/.dotnet
```

- PATH

This variable should include both the `DOTNET_ROOT` folder and the `DOTNET_ROOT/tools` folder:

Bash

```
export PATH=$PATH:$DOTNET_ROOT:$DOTNET_ROOT/tools
```

Next steps

- [How to enable TAB completion for the .NET CLI](#)
- [Tutorial: Create a console application with .NET SDK using Visual Studio Code](#)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

.NET is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

How to remove the .NET Runtime and SDK

Article • 11/29/2023

Over time, as you install updated versions of the .NET runtime and SDK, you may want to remove outdated versions of .NET from your machine. Uninstalling older versions of the runtime may change the runtime chosen to run shared framework applications, as detailed in the article on [.NET version selection](#).

Should I remove a version?

The [.NET version selection](#) behaviors and the runtime compatibility of .NET across updates enables safe removal of previous versions. .NET runtime updates are compatible within a major version **band** such as 7.x and 6.x. Additionally, newer releases of the .NET SDK generally maintain the ability to build applications that target previous versions of the runtime in a compatible manner.

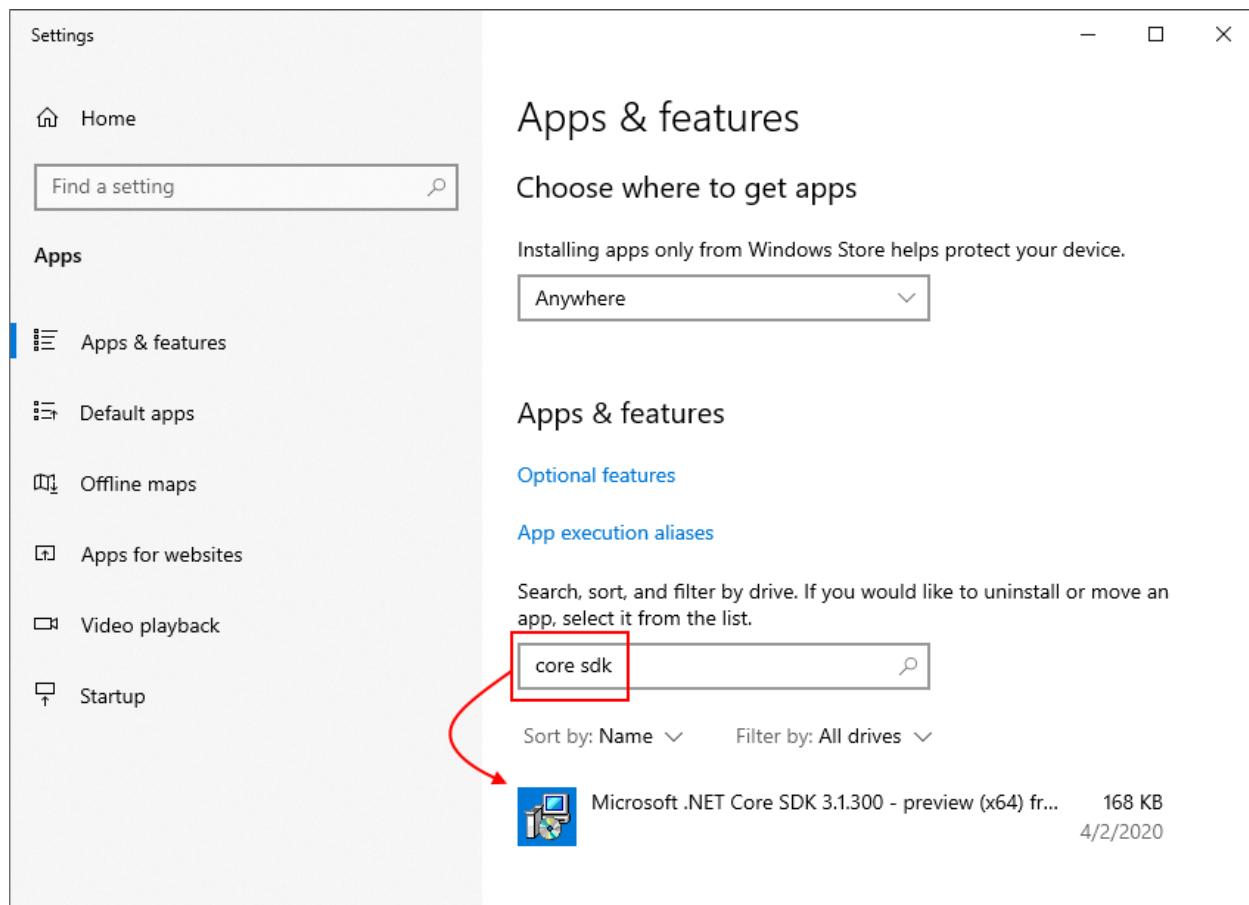
In general, you only need the latest SDK and latest patch version of the runtimes required for your application. Instances where you might want to keep older SDK or runtime versions include maintaining *project.json*-based applications. Unless your application has specific reasons for earlier SDKs or runtimes, you may safely remove older versions.

Determine what is installed

The .NET CLI has options you can use to list the versions of the SDK and runtime that are installed on your computer. Use `dotnet --list-sdks` to see the list of installed SDKs and `dotnet --list-runtimes` for the list of runtimes. For more information, see [How to check that .NET is already installed](#).

Uninstall .NET

.NET uses the Windows **Apps & features** dialog to remove versions of the .NET runtime and SDK. The following figure shows the **Apps & features** dialog. You can search for **core** or **.net** to filter and show installed versions of .NET.



Select any versions you want to remove from your computer and click **Uninstall**.

.NET Uninstall Tool

The [.NET Uninstall Tool](#) (`dotnet-core-uninstall`) lets you remove .NET SDKs and runtimes from a system. A collection of options is available to specify which versions should be uninstalled.

Visual Studio dependency on .NET SDK versions

Before Visual Studio 2019 version 16.3, Visual Studio installers called the standalone SDK installer for .NET Core version 2.1 or 2.2. As a result, the SDK versions appear in the Windows Apps & features dialog. Removing .NET SDKs that were installed by Visual Studio using the standalone installer may break Visual Studio. If Visual Studio has problems after you uninstall SDKs, run Repair on that specific version of Visual Studio. The following table shows some of the Visual Studio dependencies on .NET Core SDK versions:

| Visual Studio version | .NET Core SDK version |
|---------------------------------|--------------------------------|
| Visual Studio 2019 version 16.2 | .NET Core SDK 2.2.4xx, 2.1.8xx |

| Visual Studio version | .NET Core SDK version |
|---------------------------------|--------------------------------|
| Visual Studio 2019 version 16.1 | .NET Core SDK 2.2.3xx, 2.1.7xx |
| Visual Studio 2019 version 16.0 | .NET Core SDK 2.2.2xx, 2.1.6xx |
| Visual Studio 2017 version 15.9 | .NET Core SDK 2.2.1xx, 2.1.5xx |
| Visual Studio 2017 version 15.8 | .NET Core SDK 2.1.4xx |

Starting with Visual Studio 2019 version 16.3, Visual Studio is in charge of its own copy of the .NET SDK. For that reason, you no longer see those SDK versions in the **Apps & features** dialog.

Remove the NuGet fallback directory

Before .NET Core 3.0 SDK, the .NET Core SDK installers used a directory named *NuGetFallbackFolder* to store a cache of NuGet packages. This cache was used during operations such as `dotnet restore` or `dotnet build /t:Restore`. The *NuGetFallbackFolder* was located under the *sdk* folder where .NET is installed. For example it could be at *C:\Program Files\dotnet\sdk\NuGetFallbackFolder* on Windows and at */usr/local/share/dotnet/sdk/NuGetFallbackFolder* on macOS.

You may want to remove this directory, if:

- You're only developing using .NET Core 3.0 SDK or .NET 5 or later versions.
- You're developing using .NET Core SDK versions earlier than 3.0, but you can work online.

If you want to remove the NuGet fallback directory, you can delete it, but you'll need administrative privileges to do so.

It's not recommended to delete the *dotnet* directory. Doing so would remove any global tools you've previously installed. Also, on Windows:

- You'll break Visual Studio 2019 version 16.3 and later versions. You can run **Repair** to recover.
- If there are .NET Core SDK entries in the **Apps & features** dialog, they'll be orphaned.

 Collaborate with us on
GitHub



.NET feedback

.NET is an open source project.
Select a link to provide feedback:

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

 [Open a documentation issue](#)

 [Provide product feedback](#)

Manage .NET project and item templates

Article • 12/29/2022

.NET provides a template system that enables users to install or uninstall packages containing templates from NuGet, a NuGet package file, or a file system directory. This article describes how to manage .NET templates through the .NET SDK CLI.

For more information about creating templates, see [Tutorial: Create templates](#).

Install template

Template packages are installed through the `dotnet new install` SDK command. You can either provide the NuGet package identifier of a template package, or a folder that contains the template files.

NuGet hosted package

.NET CLI template packages are uploaded to [NuGet](#) for wide distribution. Template packages can also be installed from a private feed. Instead of uploading a template package to a NuGet feed, *nupkg* template files can be distributed and manually installed, as described in the [Local NuGet package](#) section.

For more information about configuring NuGet feeds, see [dotnet nuget add source](#).

To install a template package from the default NuGet feed, use the `dotnet new install {package-id}` command:

.NET CLI

```
dotnet new install Microsoft.DotNet.Web.Spa.ProjectTemplates
```

To install a template package from the default NuGet feed with a specific version, use the `dotnet new install {package-id}::{version}` command:

.NET CLI

```
dotnet new install Microsoft.DotNet.Web.Spa.ProjectTemplates::2.2.6
```

Local NuGet package

When a template package is created, a *nupkg* file is generated. If you have a *nupkg* file containing templates, you can install it with the `dotnet new install {path-to-package}` command:

.NET CLI

```
dotnet new install c:\code\NuGet-Packages\Some.Templates.1.0.0.nupkg
```

Folder

As an alternative to installing template from a *nupkg* file, you can also install templates from a folder directly with the `dotnet new install {folder-path}` command. The folder specified is treated as the template package identifier for any template found. Any template found in the specified folder's hierarchy is installed.

.NET CLI

```
dotnet new install c:\code\NuGet-Packages\some-folder\
```

The `{folder-path}` specified on the command becomes the template package identifier for all templates found. As specified in the [List template packages](#) section, you can get a list of template packages installed with the `dotnet new uninstall` command. In this example, the template package identifier is shown as the folder used for install:

Console

```
dotnet new uninstall
Currently installed items:

... cut to save space ...

c:\code\NuGet-Packages\some-folder
Templates:
  A Template Console Class (templateconsole) C#
  Project for some technology (contosoproject) C#
  Uninstall Command:
    dotnet new uninstall c:\code\NuGet-Packages\some-folder
```

Uninstall template package

Template packages are uninstalled through the `dotnet new uninstall` SDK command. You can either provide the NuGet package identifier of a template package, or a folder that contains the template files.

NuGet package

After a NuGet template package is installed, either from a NuGet feed or a *nupkg* file, you can uninstall it by referencing the NuGet package identifier.

To uninstall a template package, use the `dotnet new uninstall {package-id}` command:

.NET CLI

```
dotnet new uninstall Microsoft.DotNet.Web.Spa.ProjectTemplates
```

Folder

When templates are installed through a [folder path](#), the folder path becomes the template package identifier.

To uninstall a template package, use the `dotnet new uninstall {package-folder-path}` command:

.NET CLI

```
dotnet new uninstall c:\code\NuGet-Packages\some-folder
```

List template packages

By using the standard `uninstall` command without a package identifier, you can see a list of installed template packages along with the command that uninstalls each template package.

Console

```
dotnet new uninstall
Currently installed items:

... cut to save space ...

c:\code\NuGet-Packages\some-folder
Templates:
  A Template Console Class (templateconsole) C#
```

```
Project for some technology (contosoproject) C#
Uninstall Command:
dotnet new uninstall c:\code\NuGet-Packages\some-folder
```

Install template packages from other SDKs

If you've installed each version of the SDK sequentially, for example you installed SDK 6.0, then SDK 7.0, and so on, you'll have every SDK's templates installed. However, if you start with a later SDK version, like 7.0, only the templates for this version are included. Templates for any other release aren't included.

The .NET templates are available on NuGet, and you can install them like any other template. For more information, see [Install NuGet hosted package](#).

| SDK | NuGet Package Identifier |
|------------------|--|
| .NET Core 2.1 | Microsoft.DotNet.Common.ProjectTemplates.2.1 |
| .NET Core 2.2 | Microsoft.DotNet.Common.ProjectTemplates.2.2 |
| .NET Core 3.0 | Microsoft.DotNet.Common.ProjectTemplates.3.0 |
| .NET Core 3.1 | Microsoft.DotNet.Common.ProjectTemplates.3.1 |
| .NET 5.0 | Microsoft.DotNet.Common.ProjectTemplates.5.0 |
| .NET 6.0 | Microsoft.DotNet.Common.ProjectTemplates.6.0 |
| .NET 7.0 | Microsoft.DotNet.Common.ProjectTemplates.7.0 |
| ASP.NET Core 2.1 | Microsoft.DotNet.Web.ProjectTemplates.2.1 |
| ASP.NET Core 2.2 | Microsoft.DotNet.Web.ProjectTemplates.2.2 |
| ASP.NET Core 3.0 | Microsoft.DotNet.Web.ProjectTemplates.3.0 |
| ASP.NET Core 3.1 | Microsoft.DotNet.Web.ProjectTemplates.3.1 |
| ASP.NET Core 5.0 | Microsoft.DotNet.Web.ProjectTemplates.5.0 |
| ASP.NET Core 6.0 | Microsoft.DotNet.Web.ProjectTemplates.6.0 |
| ASP.NET Core 7.0 | Microsoft.DotNet.Web.ProjectTemplates.7.0 |

For example, the .NET 7 SDK includes templates for a console app targeting .NET 7. If you wanted to target .NET Core 3.1, you would need to install the 3.1 template package.

1. Try creating an app that targets .NET Core 3.1.

.NET CLI

```
dotnet new console --framework netcoreapp3.1
```

If you see an error message, you need to install the templates.

2. Install the .NET Core 3.1 project templates.

.NET CLI

```
dotnet new install Microsoft.DotNet.Common.ProjectTemplates.3.1
```

3. Try creating the app a second time.

.NET CLI

```
dotnet new console --framework netcoreapp3.1
```

And you should see a message indicating the project was created.

The template "Console Application" was created successfully.

Processing post-creation actions... Running 'dotnet restore' on path-to-project-file.csproj... Determining projects to restore... Restore completed in 1.05 sec for path-to-project-file.csproj.

Restore succeeded.

See also

- [Tutorial: Create an item template](#)
- [dotnet new](#)
- [dotnet nuget add source](#)

 Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

more information, see [our contributor guide](#).

macOS Catalina Notarization and the impact on .NET downloads and projects

Article • 05/20/2022

Beginning with macOS Catalina (version 10.15), all software built after June 1, 2019, and distributed with Developer ID, must be notarized. This requirement applies to the .NET runtime, .NET SDK, and software created with .NET. This article describes the common scenarios you may face with .NET and macOS notarization.

Installing .NET

The installers for .NET (both runtime and SDK) have been notarized since February 18, 2020. Prior released versions aren't notarized. You can manually install a non-notarized version of .NET by first downloading the installer, and then using the `sudo installer` command. For more information, see [Download and manually install for macOS](#).

Native appHost

In .NET SDK 7 and later versions, an **appHost**, which is a native Mach-O executable, is produced for your app. This executable is usually invoked by .NET when your project compiles, publishes, or is run with the `dotnet run` command. The non-**appHost** version of your app is a *dll* file that can be invoked by the `dotnet <app.dll>` command.

When run locally, the SDK signs the apphost using [ad hoc signing](#), which allows the app to run locally. When distributing your app, you'll need to properly sign your app according to Apple guidance.

You can also distribute your app without the apphost and rely on users to run your app using `dotnet`. To turn off **appHost** generation, add the `UseAppHost` boolean setting in the project file and set it to `false`. You can also toggle the appHost with the `-p:UseAppHost` parameter on the command line for the specific `dotnet` command you run:

- Project file

XML

```
<PropertyGroup>
  <UseAppHost>false</UseAppHost>
```

```
</PropertyGroup>
```

- Command-line parameter

.NET CLI

```
dotnet run -p:UseAppHost=false
```

An **appHost** is required when you publish your app **self-contained** and you cannot disable it.

For more information about the `UseAppHost` setting, see [MSBuild properties for Microsoft.NET.Sdk](#).

Context of the appHost

When the **appHost** is enabled in your project, and you use the `dotnet run` command to run your app, the app is invoked in the context of the **appHost** and not the default host (the default host is the `dotnet` command). If the **appHost** is disabled in your project, the `dotnet run` command runs your app in the context of the default host. Even if the **appHost** is disabled, publishing your app as **self-contained** generates an **appHost** executable, and users use that executable to run your app. Running your app with `dotnet <filename.dll>` invokes the app with the default host, the shared runtime.

When an app using the **appHost** is invoked, the certificate partition accessed by the app is different from the notarized default host. If your app must access the certificates installed through the default host, use the `dotnet run` command to run your app from its project file, or use the `dotnet <filename.dll>` command to start the app directly.

More information about this scenario is provided in the [ASP.NET Core and macOS and certificates](#) section.

ASP.NET Core, macOS, and certificates

.NET provides the ability to manage certificates in the macOS Keychain with the [System.Security.Cryptography.X509Certificates](#) class. Access to the macOS Keychain uses the applications identity as the primary key when deciding which partition to consider. For example, unsigned applications store secrets in the unsigned partition, but signed applications store their secrets in partitions only they can access. The source of execution that invokes your app decides which partition to use.

.NET provides three sources of execution: [appHost](#), default host (the `dotnet` command), and a custom host. Each execution model may have different identities, either signed or unsigned, and has access to different partitions within the Keychain. Certificates imported by one mode may not be accessible from another. For example, the notarized versions of .NET have a default host that is signed. Certificates are imported into a secure partition based on its identity. These certificates aren't accessible from a generated appHost, as the appHost is ad-hoc signed.

Another example, by default, ASP.NET Core imports a default SSL certificate through the default host. ASP.NET Core applications that use an appHost won't have access to this certificate and will receive an error when .NET detects the certificate isn't accessible. The error message provides instructions on how to fix this problem.

If certificate sharing is required, macOS provides configuration options with the `security` utility.

For more information on how to troubleshoot ASP.NET Core certificate issues, see [Enforce HTTPS in ASP.NET Core](#).

Default entitlements

.NET's default host (the `dotnet` command) has a set of default entitlements. These entitlements are required for proper operation of .NET. It's possible that your application may need additional entitlements, in which case you'll need to generate and use an [appHost](#) and then add the necessary entitlements locally.

Default set of entitlements for .NET:

- `com.apple.security.cs.allow-jit`
- `com.apple.security.cs.allow-unsigned-executable-memory`
- `com.apple.security.cs.allow-dyld-environment-variables`
- `com.apple.security.cs.disable-library-validation`

Notarize a .NET app

If you want your application to run on macOS Catalina (version 10.15) or higher, you'll want to notarize your app. The appHost you submit with your application for notarization should be used with at least the same [default entitlements](#) for .NET Core.

Next steps

- Install .NET on macOS.

 **Collaborate with us on GitHub**

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

.NET is an open source project.
Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

Troubleshoot .NET errors related to missing files on Linux

Article • 05/16/2023

When you try to use .NET on Linux, commands such as `dotnet new` and `dotnet run` may fail with a message related to a file not being found, such as *fxr*, *libhostfxr.so*, or *FrameworkList.xml*. Some of the error messages may be similar to the following items:

- **System.IO.FileNotFoundException**

System.IO.FileNotFoundException: Could not find file
'/usr/share/dotnet/packs/Microsoft.NETCore.App.Ref/5.0.0/data/FrameworkList
.xml'.

- **A fatal error occurred.**

A fatal error occurred. The required library *libhostfxr.so* could not be found.

or

A fatal error occurred. The folder [/usr/share/dotnet/host/fxr] does not exist.

or

A fatal error occurred, the folder [/usr/share/dotnet/host/fxr] does not contain any version-numbered child folders.

- **Generic messages about dotnet not found**

A general message may appear that indicates the SDK isn't found, or that the package has already been installed.

One symptom of these problems is that both the `/usr/lib64/dotnet` and `/usr/share/dotnet` folders are on your system.

💡 Tip

Use the `dotnet --info` command to list which SDKs and Runtimes are installed. For more information, see [How to check that .NET is already installed](#).

What's going on

These errors usually occur when two Linux package repositories provide .NET packages. While Microsoft provides a Linux package repository to source .NET packages, some Linux distributions also provide .NET packages. These distributions include:

- Alpine Linux
- Arch
- CentOS
- CentOS Stream
- Fedora
- RHEL
- Ubuntu 22.04+

If you mix .NET packages from two different sources, you'll likely run into problems. The packages might place things at different paths and might be compiled differently.

Solutions

The solution to these problems is to use .NET from one package repository. Which repository to pick, and how to do it, varies by use-case and the Linux distribution.

- [My Linux distribution provides .NET packages, and I want to use them.](#)
- [I need a version of .NET that isn't provided by my Linux distribution.](#)

My Linux distribution provides .NET packages, and I want to use them

- Do you use the Microsoft repository for other packages, such as PowerShell and MSSQL?
 - Yes

Configure your package manager to ignore the .NET packages from the Microsoft repository. It's possible that you've installed .NET from both repositories, so you want to choose one or the other.

1. Remove the existing .NET packages from your distribution. You want to start over and ensure that you don't install them from the wrong repository.

```
sudo dnf remove 'dotnet*' 'aspnet*' 'netstandard*'
```

2. Configure the Microsoft repository to ignore .NET packages.

Bash

```
echo 'excludepkgs=dotnet*,aspnet*,netstandard*' | sudo tee -a /etc/yum.repos.d/microsoft-prod.repo
```

3. Reinstall .NET from the distribution's package feed. For more information, see [Install .NET on Linux](#).

- o No

1. Remove the existing .NET packages from your distribution. You want to start over and ensure that you don't install them from the wrong repository.

Bash

```
sudo dnf remove 'dotnet*' 'aspnet*' 'netstandard*'
```

2. Delete the Microsoft repository feed from your distribution.

Bash

```
sudo dnf remove packages-microsoft-prod
```

3. Reinstall .NET from the distribution's package feed. For more information, see [Install .NET on Linux](#).

I need a version of .NET that isn't provided by my Linux distribution

Configure your package manager to ignore the .NET packages from the distribution's repository. It's possible that you've installed .NET from both repositories, so you want to choose one or the other.

1. Remove the existing .NET packages from your distribution. You want to start over and ensure that you don't install them from the wrong repository.

Bash

```
sudo dnf remove 'dotnet*' 'aspnet*' 'netstandard*'
```

2. Configure the Linux repository to ignore .NET packages.

Bash

```
echo 'excludepkgs=dotnet*,aspnet*,netstandard*' | sudo tee -a  
/etc/yum.repos.d/<your-package-source>.repo
```

Make sure to replace `<your-package-source>` with your distribution's package source.

3. Reinstall .NET from the distribution's package feed. For more information, see [Install .NET on Linux](#).

Online references

Many other users have reported these problems. The following is a list of those issues. You can read through them for insights on what may be happening:

- System.IO.FileNotFoundException and
`'/usr/share/dotnet/packs/Microsoft.NETCore.App.Ref/5.0.0/data/FrameworkList.xml'`
 - [SDK #15785: unable to build brand new project after upgrading to 5.0.3 ↗](#)
 - [SDK #15863: "MSB4018 ResolveTargetingPackAssets task failed unexpectedly" after updating to 5.0.103 ↗](#)
 - [SDK #17411: dotnet build always throwing error ↗](#)
 - [SDK #12075: dotnet 3.1.301 on Fedora 32 unable to find FrameworkList.xml because it doesn't exist ↗](#)
- Fatal error: `libhostfxr.so` couldn't be found
 - [SDK #17570: After updating Fedora 33 to 34 and dotnet 5.0.5 to 5.0.6 I get error regarding libhostfxr.so ↗](#)
- Fatal error: folder `/host/fxr` doesn't exist
 - [Core #5746: The folder does not exist when installing 3.1 on CentOS 8 with packages.microsoft.com repo enabled ↗](#)
 - [SDK #15476: A fatal error occurred. The folder '/usr/share/dotnet/host/fxr' does not exist ↗](#)
- Fatal error: folder `/host/fxr` doesn't contain any version-numbered child folders

- [Installer #9254: Error when install dotnet/core/aspnet:3.1 on CentOS 8 - Folder does not contain any version-numbered child folders ↗](#)
- [StackOverflow: Error when install dotnet/core/aspnet:3.1 on CentOS 8 - Folder does not contain any version-numbered child folders ↗](#)
- Generic errors without clear messages
 - [Core #4605: cannot run "dotnet new console" ↗](#)
 - [Core #4644: Cannot install .NET Core SDK 2.1 on Fedora 32 ↗](#)
 - [Runtime #49375: After updating to 5.0.200-1 using package manager, it appears that no sdks are installed ↗](#)

See also

- [How to check that .NET is already installed](#)
- [How to remove the .NET Runtime and SDK](#)
- [Install .NET on Linux](#)

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



.NET feedback

.NET is an open source project.
Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

How to check that .NET is already installed

Article • 12/30/2023

This article teaches you how to check which versions of the .NET runtime and SDK are installed on your computer. If you have an integrated development environment, such as Visual Studio, .NET may have already been installed.

Installing an SDK installs the corresponding runtime.

If any command in this article fails, you don't have the runtime or SDK installed. For more information, see the install articles for [Windows](#), [macOS](#), or [Linux](#).

Check SDK versions

You can see which versions of the .NET SDK are currently installed with a terminal. Open a terminal and run the following command.

.NET CLI

```
dotnet --list-sdks
```

You get output similar to the following.

Console

```
3.1.424 [C:\program files\dotnet\sdk]
5.0.100 [C:\program files\dotnet\sdk]
6.0.402 [C:\program files\dotnet\sdk]
7.0.404 [C:\program files\dotnet\sdk]
8.0.100 [C:\program files\dotnet\sdk]
```

Check runtime versions

You can see which versions of the .NET runtime are currently installed with the following command.

.NET CLI

```
dotnet --list-runtimes
```

You get output similar to the following.

Console

```
Microsoft.AspNetCore.App 3.1.30 [C:\Program  
Files\dotnet\shared\Microsoft.AspNetCore.App]  
Microsoft.AspNetCore.App 6.0.10 [C:\Program  
Files\dotnet\shared\Microsoft.AspNetCore.App]  
Microsoft.AspNetCore.App 7.0.5 [C:\Program  
Files\dotnet\shared\Microsoft.AspNetCore.App]  
Microsoft.AspNetCore.App 8.0.0 [C:\Program  
Files\dotnet\shared\Microsoft.AspNetCore.App]  
Microsoft.NETCore.App 3.1.30 [C:\Program  
Files\dotnet\shared\Microsoft.NETCore.App]  
Microsoft.NETCore.App 5.0.17 [C:\Program  
Files\dotnet\shared\Microsoft.NETCore.App]  
Microsoft.NETCore.App 6.0.10 [C:\Program  
Files\dotnet\shared\Microsoft.NETCore.App]  
Microsoft.NETCore.App 7.0.5 [C:\Program  
Files\dotnet\shared\Microsoft.NETCore.App]  
Microsoft.NETCore.App 8.0.0 [C:\Program  
Files\dotnet\shared\Microsoft.NETCore.App]  
Microsoft.WindowsDesktop.App 3.1.30 [C:\Program  
Files\dotnet\shared\Microsoft.WindowsDesktop.App]  
Microsoft.WindowsDesktop.App 6.0.10 [C:\Program  
Files\dotnet\shared\Microsoft.WindowsDesktop.App]  
Microsoft.WindowsDesktop.App 7.0.5 [C:\Program  
Files\dotnet\shared\Microsoft.WindowsDesktop.App]  
Microsoft.WindowsDesktop.App 8.0.0 [C:\Program  
Files\dotnet\shared\Microsoft.WindowsDesktop.App]
```

Check for install folders

It's possible that .NET is installed but not added to the `PATH` variable for your operating system or user profile. In this case, the commands from the previous sections may not work. As an alternative, you can check that the .NET install folders exist.

When you install .NET from an installer or script, it's installed to a standard folder. Much of the time the installer or script you're using to install .NET gives you an option to install to a different folder. If you choose to install to a different folder, adjust the start of the folder path.

- **dotnet executable**

`C:\program files\dotnet\dotnet.exe`

- **.NET SDK**

`C:\program files\dotnet\sdk\{version}\`

- .NET Runtime

`C:\program files\dotnet\shared\{runtime-type}\{version}\`

More information

You can see both the SDK versions and runtime versions with the command `dotnet --info`. You'll also get other environmental related information, such as the operating system version and runtime identifier (RID).

Next steps

- [Install the .NET Runtime and SDK for Windows.](#)
- [Install the .NET Runtime and SDK for macOS.](#)
- [Install the .NET Runtime and SDK for Linux.](#)

See also

- [Determine which .NET Framework versions are installed](#)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



.NET feedback

.NET is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

How to install localized IntelliSense files for .NET

Article • 12/29/2022

IntelliSense is a code-completion aid that's available in different integrated development environments (IDEs), such as Visual Studio. By default, when you're developing .NET projects, the SDK only includes the English version of the IntelliSense files. This article explains:

- How to install the localized version of those files.
- How to modify the Visual Studio installation to use a different language.

Note

Localized IntelliSense files are no longer available. The latest version they're available for is .NET 5. We recommend using the English IntelliSense files.

Prerequisites

- [.NET SDK](#).
- [Visual Studio 2019 version 16.3](#) or a later version.

Download and install the localized IntelliSense files

Important

This procedure requires that you have administrator permission to copy the IntelliSense files to the .NET installation folder.

1. Go to the [Download IntelliSense files](#) page.
2. Download the IntelliSense file for the language and version you'd like to use.
3. Extract the contents of the zip file.
4. Navigate to the .NET Intellisense folder.

- a. Navigate to the .NET installation folder. By default, it's under `%ProgramFiles%\dotnet\packs`.
- b. Choose which SDK you want to install the IntelliSense for, and navigate to the associated path. You have the following options:

| SDK type | Path |
|------------------|---|
| .NET 6 and later | <code>Microsoft.NETCore.App.Ref</code> |
| Windows Desktop | <code>Microsoft.WindowsDesktop.App.Ref</code> |
| .NET Standard | <code>NETStandard.Library.Ref</code> |

- c. Navigate to the version you want to install the localized IntelliSense for. For example, `5.0.0`.
- d. Open the `ref` folder.
- e. Open the moniker folder. For example, `net5.0`.

So, the full path that you'd navigate to would look similar to `C:\Program Files\dotnet\packs\Microsoft.NETCore.App.Ref\5.0.0\ref\net5.0`.

5. Create a subfolder inside the moniker folder you just opened. The name of the folder indicates which language you want to use. The following table specifies the different options:

| Language | Folder name |
|-----------------------|----------------------|
| Brazilian Portuguese | <code>pt-br</code> |
| Chinese (simplified) | <code>zh-hans</code> |
| Chinese (traditional) | <code>zh-hant</code> |
| French | <code>fr</code> |
| German | <code>de</code> |
| Italian | <code>it</code> |
| Japanese | <code>ja</code> |
| Korean | <code>ko</code> |
| Russian | <code>ru</code> |
| Spanish | <code>es</code> |

6. Copy the *.xml* files you extracted in step 3 to this new folder. The *.xml* files are broken down by SDK folders, so copy them to the matching SDK you chose in step 4.

Modify Visual Studio language

For Visual Studio to use a different language for IntelliSense, install the appropriate language pack. This can be done [during installation](#) or at a later time by modifying the Visual Studio installation. If you already have Visual Studio configured to the language of your choice, your IntelliSense installation is ready.

Install the language pack

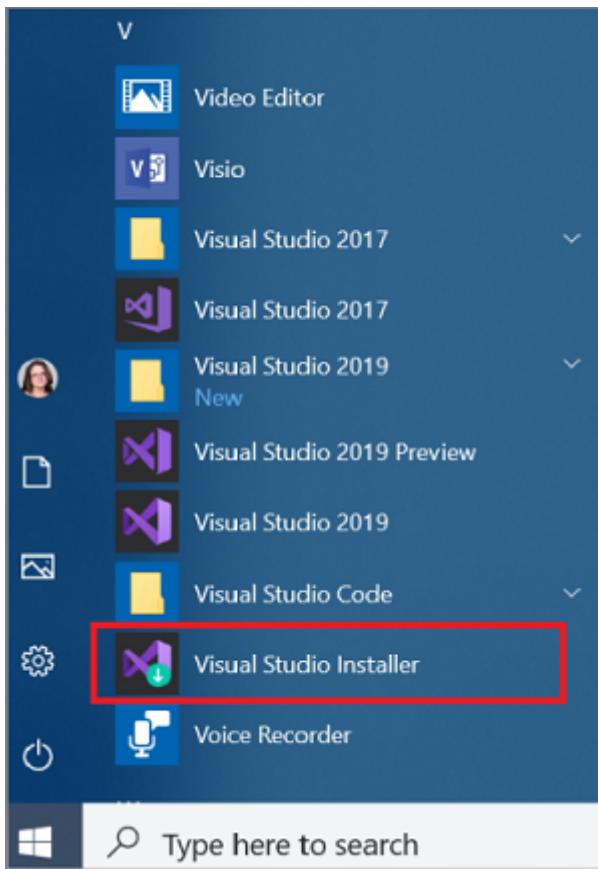
If you didn't install the desired language pack during setup, update Visual Studio as follows to install the language pack:

 **Important**

To install, update, or modify Visual Studio, you must log on with an account that has administrator permission. For more information, see [User permissions and Visual Studio](#).

1. Find the Visual Studio Installer on your computer.

For example, on a computer running Windows 10, select **Start**, and then scroll to the letter **V**, where it's listed as **Visual Studio Installer**.



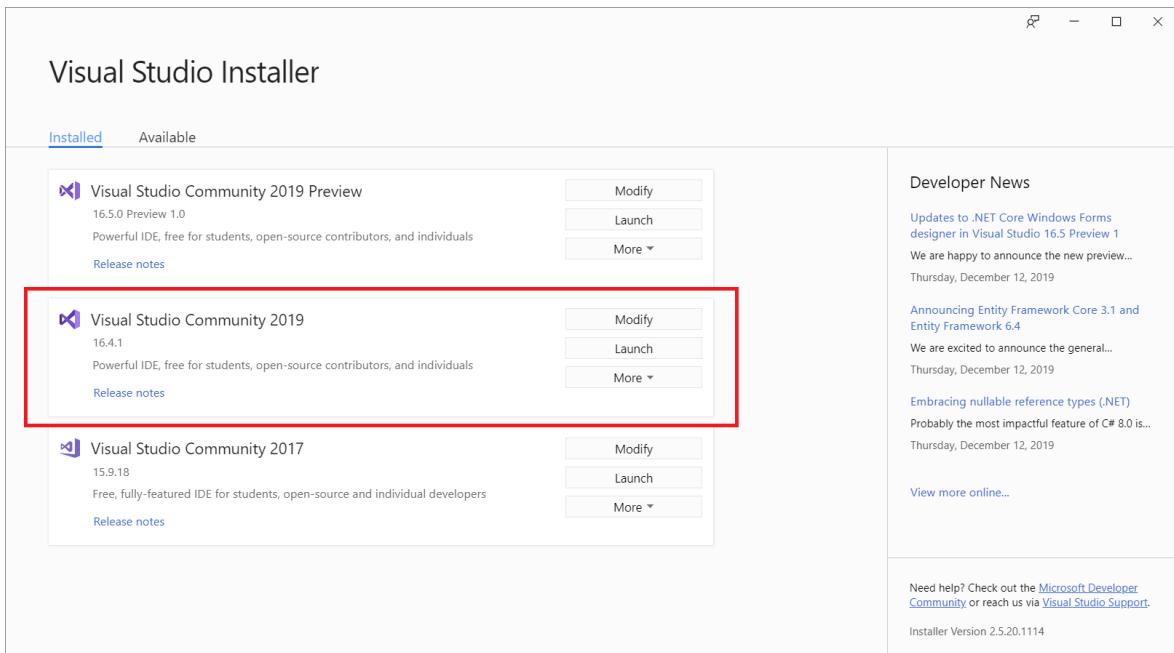
ⓘ Note

You can also find the Visual Studio Installer in the following location:

```
C:\Program Files (x86)\Microsoft Visual Studio\Installer\vs_installer.exe
```

You might have to update the installer before continuing. If so, follow the prompts.

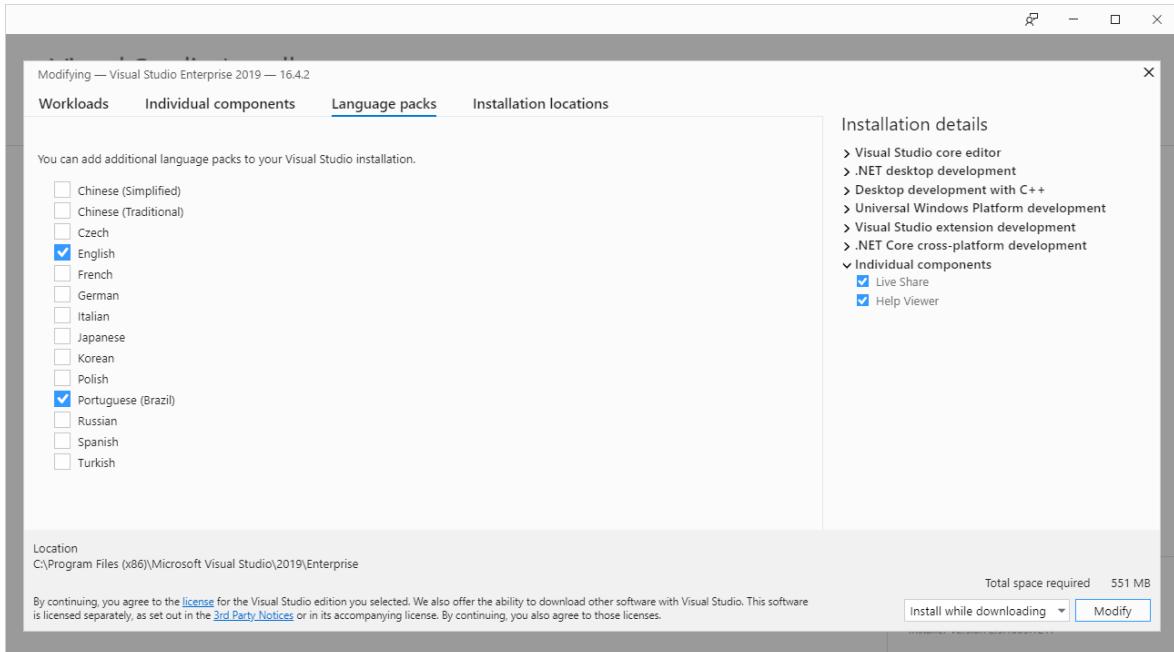
2. In the installer, look for the edition of Visual Studio that you want to add the language pack to, and then choose **Modify**.



ⓘ Important

If you don't see a **Modify** button but you see an **Update** one instead, you need to update your Visual Studio before you can modify your installation. After the update is finished, the **Modify** button should appear.

3. In the **Language packs** tab, select or deselect the languages you want to install or uninstall.



4. Choose **Modify**. The update starts.

Modify language settings in Visual Studio

Once you've installed the desired language packs, modify your Visual Studio settings to use a different language:

1. Open Visual Studio.
2. On the start window, choose **Continue without code**.
3. On the menu bar, select **Tools > Options**. The Options dialog opens.
4. Under the **Environment** node, choose **International Settings**.
5. On the **Language** drop-down, select the desired language. Choose **OK**.
6. A dialog informs you that you have to restart Visual Studio for the changes to take effect. Choose **OK**.
7. Restart Visual Studio.

After this, your IntelliSense should work as expected when you open a .NET project that targets the version of the IntelliSense files you just installed.

See also

- [IntelliSense in Visual Studio](#)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



.NET feedback

The .NET documentation is open source. Provide feedback here.

[Open a documentation issue](#)

[Provide product feedback](#)

Introduction to .NET

Article • 01/10/2024

.NET is a free, cross-platform, [open-source developer platform](#) for building [many kinds of applications](#). It can run programs written in [multiple languages](#), with [C#](#) being the most popular. It relies on a [high-performance](#) runtime that is used in production by many [high-scale apps](#).

To learn how to [download .NET](#) and start writing your first app, see [Getting started](#).

The .NET platform has been designed to deliver productivity, performance, security, and reliability. It provides automatic memory management via a [garbage collector \(GC\)](#). It is type-safe and memory-safe, due to using a GC and strict language compilers. It offers [concurrency](#) via `async/await` and `Task` primitives. It includes a large set of libraries that have broad functionality and have been optimized for performance on multiple operating systems and chip architectures.

.NET has the following [design points](#):

- **Productivity is full-stack** with runtime, libraries, language, and tools all contributing to developer user experience.
- **Safe code** is the primary compute model, while [unsafe code](#) enables additional manual optimizations.
- **Static and dynamic code** are both supported, enabling a broad set of distinct scenarios.
- **Native code interop and hardware intrinsics** are low cost and high-fidelity (raw API and instruction access).
- **Code is portable across platforms** (OS and chip architecture), while platform targeting enables specialization and optimization.
- **Adaptability across programming domains** (cloud, client, gaming) is enabled with specialized implementations of the general-purpose programming model.
- **Industry standards** like OpenTelemetry and gRPC are favored over bespoke solutions.

.NET is maintained by Microsoft and the community. It is regularly updated to ensure users deploy secure and reliable applications to production.

Components

.NET includes the following components:

- Runtime -- executes application code.
- Libraries -- provides utility functionality like [JSON parsing](#).
- Compiler -- compiles C# (and other languages) source code into (runtime) executable code.
- SDK and other tools -- enable building and monitoring apps with modern workflows.
- App stacks -- like ASP.NET Core and Windows Forms, that enable writing apps.

The runtime, libraries, and languages are the pillars of the .NET stack. Higher-level components, like .NET tools, and app stacks, like ASP.NET Core, build on top of these pillars. C# is the primary programming language for .NET and much of .NET is written in C#.

C# is object-oriented and the runtime supports object orientation. C# requires garbage collection and the runtime provides a tracing garbage collector. The libraries (and also the app stacks) shape those capabilities into concepts and object models that enable developers to productively write algorithms in intuitive workflows.

The core libraries expose thousands of types, many of which integrate with and fuel the C# language. For example, C#'s `foreach` statement lets you enumerate arbitrary collections. Pattern-based optimizations enable collections like `List<T>` to be processed simply and efficiently. You can leave resource management up to garbage collection, but prompt cleanup is possible via `IDisposable` and direct language support in the `using` statement.

Support for doing multiple things at the same time is fundamental to practically all workloads. That could be client applications doing background processing while keeping the UI responsive, services handling many thousands of simultaneous requests, devices responding to a multitude of simultaneous stimuli, or high-powered machines parallelizing the processing of compute-intensive operations. Asynchronous programming support is a first-class feature of the C# programming language, which provides the `async` and `await` keywords that make it easy to write and compose asynchronous operations while still enjoying the full benefits of all the control flow constructs the language has to offer.

The [type system](#) offers significant breadth, catering somewhat equally to safety, descriptiveness, dynamism, and native interop. First and foremost, the type system enables an object-oriented programming model. It includes types, (single base class) inheritance, interfaces (including default method implementations), and virtual method dispatch to provide a sensible behavior for all the type layering that object orientation allows. [Generic types](#) are a pervasive feature that let you specialize classes to one or more types.

The .NET runtime provides automatic memory management via a garbage collector. For any language, its memory management model is likely its most defining characteristic. This is true for .NET languages. .NET has a self-tuning, tracing GC. It aims to deliver "hands off" operation in the general case while offering configuration options for more extreme workloads. The current GC is the result of many years of investment and learnings from a multitude of workloads.

Value types and stack-allocated memory blocks offer more direct, low-level control over data and native platform interop, in contrast to .NET's GC-managed types. Most of the primitive types in .NET, like integer types, are value types, and users can define their own types with similar semantics. Value types are fully supported through .NET's generics system, meaning that generic types like `List<T>` can provide flat, no-overhead memory representations of value type collections.

[Reflection](#) is a "programs as data" paradigm, allowing one part of a program to dynamically query and invoke another, in terms of assemblies, types, and members. It's particularly useful for late-bound programming models and tools.

Exceptions are the primary error handling model in .NET. Exceptions have the benefit that error information does not need to be represented in method signatures or handled by every method. Proper exception handling is essential for application reliability. To prevent your app from crashing, you can intentionally handle expected exceptions in your code. A crashed app is more reliable and diagnosable than an app with undefined behavior.

App stacks, like ASP.NET Core and Windows Forms, build on and take advantage of low-level libraries, language, and runtime. The app stacks define the way that apps are constructed and their lifecycle of execution.

The SDK and other tools enable a modern developer experience, both on a developer desktop and for continuous integration (CI). The modern developer experience includes being able to build, analyze, and test code. .NET projects can often be built by a single `dotnet build` command, which orchestrates restoring NuGet packages and building dependencies.

NuGet is the package manager for .NET. It contains hundreds of thousands of packages that implement functionality for many scenarios. A majority of apps rely on NuGet packages for some functionality. The [NuGet Gallery](#) is maintained by Microsoft.

Free and open source

.NET is free, open source, and is a [.NET Foundation](#) project. .NET is maintained by Microsoft and the community on GitHub in [several repositories](#).

.NET source and binaries are licensed with the [MIT license](#). Additional [licenses apply on Windows](#).

Support

.NET is [supported by multiple organizations](#) that work to ensure that .NET can run on [multiple operating systems](#) and kept up to date. It can be used on Arm64, x64, and x86 architectures.

New versions of .NET are released annually in November, per our [releases and support policies](#). It is [updated monthly](#) on Patch Tuesday (second Tuesday), typically at 10AM Pacific time.

.NET ecosystem

There are multiple variants of .NET, each supporting a different type of app. The reason for multiple variants is part historical, part technical.

.NET implementations:

- **.NET Framework** -- The original .NET. It provides access to the broad capabilities of Windows and Windows Server. It is actively supported, in maintenance.
- **Mono** -- The original community and open source .NET. A cross-platform implementation of .NET Framework. Actively supported for Android, iOS, and WebAssembly.
- **.NET (Core)** -- Modern .NET. A cross-platform and open source implementation of .NET, rethought for the cloud age while remaining significantly compatible with .NET Framework. Actively supported for Linux, macOS, and Windows.

Next steps

- [Choose a .NET tutorial](#)
- [Try .NET in your browser](#)
- [Take a tour of C#](#)
- [Take a tour of F#](#)

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

.NET is an open source project.
Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

Build apps with .NET

Article • 01/10/2024

.NET has support for building many kinds of apps, including client, cloud, and gaming.

Cloud apps

- [.NET Aspire](#)
- [Serverless functions](#)
- [Web and microservices](#)

Client apps

- [Mobile](#)
- [Games ↗](#)
- [Desktop apps](#)

Other app types

- [Console apps](#)
- [Internet of Things \(IoT\)](#)
- [Machine learning](#)
- [Windows services](#)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



.NET feedback

.NET is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

Microsoft .NET language strategy

Article • 02/06/2023

Microsoft offers 3 languages on the .NET platform – C#, F# and Visual Basic. In this article, you'll learn about our strategy for each language. Look for links to additional articles on how these strategies guide us and ways to learn more about each language.

C#

C# is a cross-platform general purpose language that makes developers productive while writing highly performant code. With millions of developers, C# is the most popular .NET language. C# has broad support in the ecosystem and all .NET [workloads](#). Based on object-oriented principles, it incorporates many features from other paradigms, not least functional programming. Low-level features support high-efficiency scenarios without writing unsafe code. Most of the .NET runtime and libraries are written in C#, and advances in C# often benefit all .NET developers.

Our strategy for C#

We will keep evolving C# to meet the changing needs of developers and remain a state-of-the-art programming language. We will innovate eagerly and broadly in collaboration with the teams responsible for .NET libraries, developer tools, and workload support, while being careful to stay within the spirit of the language. Recognizing the diversity of domains where C# is being used, we will prefer language and performance improvements that benefit all or most developers and maintain a high commitment to backwards compatibility. We will continue to empower the broader .NET ecosystem and grow its role in C#'s future, while maintaining stewardship of design decisions.

You can read more about how this strategy guides us in the [C# guide](#).

F#

F# is a succinct, robust and performant language that is expression-based and immutable by default. It focuses on expressive power, simplicity and elegance and is used by many thousands of developers that appreciate its pragmatic function-first approach to .NET. F# offers the full power of .NET and works well with C# for mixed language solutions. The community makes significant contributions to the compiler and runtime, as well as a broad array of F# tools and frameworks.

Our strategy for F#

We will drive F# evolution and support the F# ecosystem with language leadership and governance. We will encourage community contributions to improve the F# language and developer experience. We will continue to rely on the community to provide important libraries, developer tools and [workload](#) support. As the language evolves, F# will support .NET platform improvements and maintain interoperability with new C# features. We will work across language, tooling, and documentation to lower the barrier to entry into F# for new developers and organizations as well as broadening its reach into new domains.

You can read more about how this strategy guides us in the [F# guide](#).

Visual Basic

Visual Basic (VB) has a long history as an approachable language favoring clarity over brevity. Its hundreds of thousands of developers are concentrated around the traditional Windows-based client [workloads](#) where VB has long pioneered great tooling and ease of use. Today's VB developers benefit from a stable and mature object-oriented language paired with a growing .NET ecosystem and ongoing tooling improvements. Some .NET workloads are not supported in VB, and it is common for VB developers to use C# for those scenarios.

Our strategy for Visual Basic

We will ensure Visual Basic remains a straightforward and approachable language with a stable design. The core libraries of .NET (such as the BCL) will support VB and many of the improvements to the .NET Runtime and libraries will automatically benefit VB. When C# or the .NET Runtime introduce new features that would require language support, VB will generally adopt a consumption-only approach and avoid new syntax. We do not plan to extend Visual Basic to new workloads. We will continue to invest in the experience in Visual Studio and interop with C#, especially in core VB scenarios such as Windows Forms and libraries.

You can read more about how this strategy guides us in the [Visual Basic guide](#).

 Collaborate with us on
GitHub



.NET feedback

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

.NET implementations

Article • 11/19/2023

A .NET app is developed for one or more *implementations* of .NET. Implementations of .NET include .NET Framework, .NET 5+ (and .NET Core), and Mono.

Each implementation of .NET includes the following components:

- One or more runtimes—for example, .NET Framework CLR and .NET 8 CLR.
- A class library—for example, .NET Framework Base Class Library and .NET 8 Base Class Library.
- Optionally, one or more application frameworks—for example, [ASP.NET](#), [Windows Forms](#), and [Windows Presentation Foundation \(WPF\)](#) are included in .NET Framework and .NET 5+.
- Optionally, development tools. Some development tools are shared among multiple implementations.

There are four .NET implementations that Microsoft supports:

- .NET 6 and later versions
- .NET Framework
- Mono
- UWP

.NET, previously referred to as .NET Core, is currently the primary implementation. .NET (8) is built on a single code base that supports multiple platforms and many workloads, such as Windows desktop apps and cross-platform console apps, cloud services, and websites. [Some workloads](#), such as .NET WebAssembly build tools, are available as optional installations.

.NET 5 and later versions

.NET, previously referred to as .NET Core, is a cross-platform implementation of .NET that's designed to handle server and cloud workloads at scale. It also supports other workloads, including desktop apps. It runs on Windows, macOS, and Linux. It implements .NET Standard, so code that targets .NET Standard can run on .NET. [ASP.NET Core](#), [Windows Forms](#), and [Windows Presentation Foundation \(WPF\)](#) all run on .NET.

.NET 8 is the latest version of this .NET implementation.

For more information, see the following resources:

- [.NET introduction](#)
- [.NET vs. .NET Framework for server apps](#)
- [.NET 5+ and .NET Standard](#)

.NET Framework

.NET Framework is the original .NET implementation that has existed since 2002. Versions 4.5 and later implement .NET Standard, so code that targets .NET Standard can run on those versions of .NET Framework. It contains additional Windows-specific APIs, such as APIs for Windows desktop development with Windows Forms and WPF. .NET Framework is optimized for building Windows desktop applications.

For more information, see the [.NET Framework guide](#).

Mono

Mono is a .NET implementation that is mainly used when a small runtime is required. It is the runtime that powers Xamarin applications on Android, macOS, iOS, tvOS, and watchOS and is focused primarily on a small footprint. Mono also powers games built using the Unity engine.

It supports all of the currently published .NET Standard versions.

Historically, Mono implemented the larger API of .NET Framework and emulated some of the most popular capabilities on Unix. It is sometimes used to run .NET applications that rely on those capabilities on Unix.

Mono is typically used with a just-in-time compiler, but it also features a full static compiler (ahead-of-time compilation) that is used on platforms like iOS.

For more information, see the [Mono documentation](#).

Universal Windows Platform (UWP)

UWP is an implementation of .NET that is used for building modern, touch-enabled Windows applications and software for the Internet of Things (IoT). It's designed to unify the different types of devices that you may want to target, including PCs, tablets, phones, and even the Xbox. UWP provides many services, such as a centralized app store, an execution environment (AppContainer), and a set of Windows APIs to use instead of Win32 (WinRT). Apps can be written in C++, C#, Visual Basic, and JavaScript.

For more information, see [Introduction to the Universal Windows Platform](#).

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

.NET class libraries

Article • 01/12/2022

Class libraries are the [shared library](#) concept for .NET. They enable you to componentize useful functionality into modules that can be used by multiple applications. They can also be used as a means of loading functionality that is not needed or not known at application startup. Class libraries are described using the [.NET Assembly file format](#).

There are three types of class libraries that you can use:

- **Platform-specific** class libraries have access to all the APIs in a given platform (for example, .NET Framework on Windows, Xamarin iOS), but can only be used by apps and libraries that target that platform.
- **Portable** class libraries have access to a subset of APIs, and can be used by apps and libraries that target multiple platforms.
- **.NET Standard** class libraries are a merger of the platform-specific and portable library concept into a single model that provides the best of both.

Platform-specific class libraries

Platform-specific libraries are bound to a single .NET platform (for example, .NET Framework on Windows) and can therefore take significant dependencies on a known execution environment. Such an environment exposes a known set of APIs (.NET and OS APIs) and maintains and exposes expected state (for example, Windows registry).

Developers who create platform-specific libraries can fully exploit the underlying platform. The libraries will only ever run on that given platform, making platform checks or other forms of conditional code unnecessary (modulo single sourcing code for multiple platforms).

Platform-specific libraries have been the primary class library type for the .NET Framework. Even as other .NET implementations emerged, platform-specific libraries remained the dominant library type.

Portable class libraries

Portable libraries are supported on multiple .NET implementations. They can still take dependencies on a known execution environment, however, the environment is a synthetic one that's generated by the intersection of a set of concrete .NET

implementations. Exposed APIs and platform assumptions are a subset of what would be available to a platform-specific library.

You choose a platform configuration when you create a portable library. The platform configuration is the set of platforms that you need to support (for example, .NET Framework 4.5+, Windows Phone 8.0+). The more platforms you opt to support, the fewer APIs and fewer platform assumptions you can make, the lowest common denominator. This characteristic can be confusing at first, since people often think "more is better" but find that more supported platforms results in fewer available APIs.

Many library developers have switched from producing multiple platform-specific libraries from one source (using conditional compilation directives) to portable libraries. There are [several approaches](#) for accessing platform-specific functionality within portable libraries, with bait-and-switch being the most widely accepted technique at this point.

.NET Standard class libraries

.NET Standard libraries are a replacement of the platform-specific and portable libraries concepts. They are platform-specific in the sense that they expose all functionality from the underlying platform (no synthetic platforms or platform intersections). They are portable in the sense that they work on all supporting platforms.

.NET Standard exposes a set of library *contracts*. .NET implementations must support each contract fully or not at all. Each implementation, therefore, supports a set of .NET Standard contracts. The corollary is that each .NET Standard class library is supported on the platforms that support its contract dependencies.

.NET Standard does not expose the entire functionality of .NET Framework (nor is that a goal), however, the libraries do expose many more APIs than Portable Class Libraries.

The following implementations support .NET Standard libraries:

- .NET Core
- .NET Framework
- Mono
- Universal Windows Platform (UWP)

For more information, see [.NET Standard](#).

Mono class libraries

Class libraries are supported on Mono, including the three types of libraries described previously. Mono is often viewed as a cross-platform implementation of .NET Framework. In part, this is because platform-specific .NET Framework libraries can run on the Mono runtime without modification or recompilation. This characteristic was in place before the creation of portable class libraries, so was an obvious choice to enable binary portability between .NET Framework and Mono (although it only worked in one direction).

 **Collaborate with us on GitHub**

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

 .NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

.NET Standard

Article • 11/15/2023

.NET Standard is a formal specification of .NET APIs that are available on multiple .NET implementations. The motivation behind .NET Standard was to establish greater uniformity in the .NET ecosystem. .NET 5 and later versions adopt a different approach to establishing uniformity that eliminates the need for .NET Standard in most scenarios. However, if you want to share code between .NET Framework and any other .NET implementation, such as .NET Core, your library should target .NET Standard 2.0. [No new versions of .NET Standard will be released ↗](#), but .NET 5 and all later versions will continue to support .NET Standard 2.1 and earlier.

For information about choosing between .NET 5+ and .NET Standard, see [.NET 5+ and .NET Standard](#) later in this article.

.NET Standard versions

.NET Standard is versioned. Each new version adds more APIs. When a library is built against a certain version of .NET Standard, it can run on any .NET implementation that implements that version of .NET Standard (or higher).

Targeting a higher version of .NET Standard allows a library to use more APIs but means it can only be used on more recent versions of .NET. Targeting a lower version reduces the available APIs but means the library can run in more places.

Select .NET Standard version

1.0

.NET Standard 1.0 has 7,949 of the 37,118 available APIs.

[Expand table](#)

| .NET implementation | Version support |
|---------------------|---|
| .NET and .NET Core | 1.0, 1.1, 2.0, 2.1, 2.2, 3.0, 3.1, 5.0, 6.0, 7.0, 8.0 |
| .NET Framework | 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1 |
| Mono | 4.6, 5.4, 6.4 |
| Xamarin.iOS | 10.0, 10.14, 12.16 |

| .NET implementation | Version support |
|----------------------------|---------------------------------|
| Xamarin.Mac | 3.0, 3.8, 5.16 |
| Xamarin.Android | 7.0, 8.0, 10.0 |
| Universal Windows Platform | 8.0, 8.1, 10.0, 10.0.16299, TBD |
| Unity | 2018.1 |

For more information, see [.NET Standard 1.0](#). For an interactive table, see [.NET Standard versions](#).

Which .NET Standard version to target

We recommend you target .NET Standard 2.0, unless you need to support an earlier version. Most general-purpose libraries should not need APIs outside of .NET Standard 2.0. .NET Standard 2.0 is supported by all modern platforms and is the recommended way to support multiple platforms with one target.

If you need to support .NET Standard 1.x, we recommend that you *also* target .NET Standard 2.0. .NET Standard 1.x is distributed as a granular set of NuGet packages, which creates a large package dependency graph and results in developers downloading a lot of packages when building. For more information, see [Cross-platform targeting](#) and [.NET 5+ and .NET Standard](#) later in this article.

.NET Standard versioning rules

There are two primary versioning rules:

- Additive: .NET Standard versions are logically concentric circles: higher versions incorporate all APIs from previous versions. There are no breaking changes between versions.
- Immutable: Once shipped, .NET Standard versions are frozen.

There will be no new .NET Standard versions after 2.1. For more information, see [.NET 5+](#) and [.NET Standard](#) later in this article.

Specification

The .NET Standard specification is a standardized set of APIs. The specification is maintained by .NET implementers, specifically Microsoft (includes .NET Framework, .NET

Core, and Mono) and Unity.

Official artifacts

The official specification is a set of .cs files that define the APIs that are part of the standard. The [ref directory](#) in the (now archived) [dotnet/standard repository](#) defines the .NET Standard APIs.

The [NETStandard.Library](#) metapackage ([source](#)) describes the set of libraries that define (in part) one or more .NET Standard versions.

A given component, like `System.Runtime`, describes:

- Part of .NET Standard (just its scope).
- Multiple versions of .NET Standard, for that scope.

Derivative artifacts are provided to enable more convenient reading and to enable certain developer scenarios (for example, using a compiler).

- [API list in markdown](#).
- Reference assemblies, distributed as NuGet packages and referenced by the [NETStandard.Library](#) metapackage.

Package representation

The primary distribution vehicle for the .NET Standard reference assemblies is NuGet packages. Implementations are delivered in a variety of ways, appropriate for each .NET implementation.

NuGet packages target one or more [frameworks](#). .NET Standard packages target the ".NET Standard" framework. You can target the .NET Standard framework using the `netstandard compact TFM` (for example, `netstandard1.4`). Libraries that are intended to run on multiple implementations of .NET should target this framework. For the broadest set of APIs, target `netstandard2.0` since the number of available APIs more than doubled between .NET Standard 1.6 and 2.0.

The [NETStandard.Library](#) metapackage references the complete set of NuGet packages that define .NET Standard. The most common way to target `netstandard` is by referencing this metapackage. It describes and provides access to the ~40 .NET libraries and associated APIs that define .NET Standard. You can reference additional packages that target `netstandard` to get access to additional APIs.

Versioning

The specification is not singular, but a linearly versioned set of APIs. The first version of the standard establishes a baseline set of APIs. Subsequent versions add APIs and inherit APIs defined by previous versions. There is no established provision for removing APIs from the Standard.

.NET Standard is not specific to any one .NET implementation, nor does it match the versioning scheme of any of those implementations.

As noted earlier, there will be no new .NET Standard versions after 2.1.

Target .NET Standard

You can [build .NET Standard Libraries](#) using a combination of the `netstandard` framework and the `NETStandard.Library` metapackage.

.NET Framework compatibility mode

Starting with .NET Standard 2.0, the .NET Framework compatibility mode was introduced. This compatibility mode allows .NET Standard projects to reference .NET Framework libraries as if they were compiled for .NET Standard. Referencing .NET Framework libraries doesn't work for all projects, such as libraries that use Windows Presentation Foundation (WPF) APIs.

For more information, see [.NET Framework compatibility mode](#).

.NET Standard libraries and Visual Studio

To build .NET Standard libraries in Visual Studio, make sure you have [Visual Studio 2022](#), [Visual Studio 2019](#), or Visual Studio 2017 version 15.3 or later installed on Windows, or [Visual Studio for Mac version 7.1](#) or later installed on macOS.

If you only need to consume .NET Standard 2.0 libraries in your projects, you can also do that in Visual Studio 2015. However, you need NuGet client 3.6 or higher installed. You can download the NuGet client for Visual Studio 2015 from the [NuGet downloads](#) ↗ page.

.NET 5+ and .NET Standard

.NET 5, .NET 6, .NET 7, and .NET 8 are single products with a uniform set of capabilities and APIs that can be used for Windows desktop apps and cross-platform console apps, cloud services, and websites. The .NET 8 [TFMs](#), for example, reflect this broad range of scenarios:

- `net8.0`

This TFM is for code that runs everywhere. With a few exceptions, it includes only technologies that work cross-platform. For .NET 8 code, `net8.0` replaces both `netcoreapp` and `netstandard` TFMs.

- `net8.0-windows`

This is an example of an [OS-specific TFM](#) that add OS-specific functionality to everything that `net8.0` refers to.

When to target `net8.0` vs. `netstandard`

For existing code that targets `netstandard`, there's no need to change the TFM to `net8.0` or a later TFM. .NET 8 implements .NET Standard 2.1 and earlier. The only reason to retarget from .NET Standard to .NET 8+ would be to gain access to more runtime features, language features, or APIs. For example, in order to use C# 9, you need to target .NET 5 or a later version. You can multitarget .NET 8 and .NET Standard to get access to newer features and still have your library available to other .NET implementations.

Here are some guidelines for new code for .NET 5+:

- App components

If you're using libraries to break down an application into several components, we recommend you target `net8.0`. For simplicity, it's best to keep all projects that make up your application on the same version of .NET. Then you can assume the same BCL features everywhere.

- Reusable libraries

If you're building reusable libraries that you plan to ship on NuGet, consider the trade-off between reach and available feature set. .NET Standard 2.0 is the latest version that's supported by .NET Framework, so it gives good reach with a fairly large feature set. We don't recommend targeting .NET Standard 1.x, as you'd limit the available feature set for a minimal increase in reach.

If you don't need to support .NET Framework, you could go with .NET Standard 2.1 or .NET 8. We recommend you skip .NET Standard 2.1 and go straight to .NET 8. Most widely used libraries will multi-target for both .NET Standard 2.0 and .NET 5+. Supporting .NET Standard 2.0 gives you the most reach, while supporting .NET 5+ ensures you can leverage the latest platform features for customers that are already on .NET 5+.

.NET Standard problems

Here are some problems with .NET Standard that help explain why .NET 5 and later versions are the better way to share code across platforms and workloads:

- Slowness to add new APIs

.NET Standard was created as an API set that all .NET implementations would have to support, so there was a review process for proposals to add new APIs. The goal was to standardize only APIs that could be implemented in all current and future .NET platforms. The result was that if a feature missed a particular release, you might have to wait for a couple of years before it got added to a version of the Standard. Then you'd wait even longer for the new version of .NET Standard to be widely supported.

Solution in .NET 5+: When a feature is implemented, it's already available for every .NET 5+ app and library because the code base is shared. And since there's no difference between the API specification and its implementation, you're able to take advantage of new features much quicker than with .NET Standard.

- Complex versioning

The separation of the API specification from its implementations results in complex mapping between API specification versions and implementation versions. This complexity is evident in the table shown earlier in this article and the instructions for how to interpret it.

Solution in .NET 5+: There's no separation between a .NET 5+ API specification and its implementation. The result is a simplified TFM scheme. There's one TFM prefix for all workloads: `net8.0` is used for libraries, console apps, and web apps. The only variation is a [suffix that specifies platform-specific APIs](#) for a particular platform, such as `net8.0-windows`. Thanks to this TFM naming convention, you can easily tell whether a given app can use a given library. No version number equivalents table, like the one for .NET Standard, is needed.

- Platform-unsupported exceptions at run time

.NET Standard exposes platform-specific APIs. Your code might compile without errors and appear to be portable to any platform even if it isn't portable. When it runs on a platform that doesn't have an implementation for a given API, you get run-time errors.

Solution in .NET 5+: The .NET 5+ SDKs include code analyzers that are enabled by default. The platform compatibility analyzer detects unintentional use of APIs that aren't supported on the platforms you intend to run on. For more information, see [Platform compatibility analyzer](#).

.NET Standard not deprecated

.NET Standard is still needed for libraries that can be used by multiple .NET implementations. We recommend you target .NET Standard in the following scenarios:

- Use `netstandard2.0` to share code between .NET Framework and all other implementations of .NET.
- Use `netstandard2.1` to share code between Mono, Xamarin, and .NET Core 3.x.

See also

- [.NET Standard versions \(source\)](#)
- [.NET Standard versions \(interactive UI\)](#)
- [Build a .NET Standard library](#)
- [Cross-platform targeting](#)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



.NET feedback

.NET is an open source project. Select a link to provide feedback:

-  [Open a documentation issue](#)
-  [Provide product feedback](#)

Releases and support for .NET

Article • 10/14/2023

Microsoft ships major releases, minor releases, and servicing updates (patches) for .NET. This article explains release types, servicing updates, SDK feature bands, support periods, and support options.

ⓘ Note

For information about versioning and support for .NET Framework, see [.NET Framework Lifecycle](#).

Release types

Information about the type of each release is encoded in the version number in the form *major.minor.patch*.

For example:

- .NET 6 and .NET 7 are major releases.
- .NET Core 3.1 is the first minor release after the .NET Core 3.0 major release.
- .NET Core 5.0.15 is the fifteenth patch for .NET 5.

For a list of released versions of .NET and information about how often .NET ships, see the [Support Policy](#).

Major releases

Major releases include new features, new public API surface area, and bug fixes.

Examples include .NET 6 and .NET 7. Due to the nature of the changes, these releases are expected to have breaking changes. Major releases install side by side with previous major releases.

Minor releases

Minor releases also include new features, public API surface area, and bug fixes, and may also have breaking changes. An example is .NET Core 3.1. The difference between these and major releases is that the magnitude of the changes is smaller. An application upgrading from .NET Core 3.0 to 3.1 has a smaller jump to move forward. Minor releases install side by side with previous minor releases.

Servicing updates

Servicing updates (patches) ship almost every month, and these updates carry both security and non-security bug fixes. For example, .NET 5.0.8 was the eighth update for .NET 5. When these updates include security fixes, they're released on "patch Tuesday", which is always the second Tuesday of the month. Servicing updates are expected to maintain compatibility. Starting with .NET Core 3.1, servicing updates are upgrades that remove the preceding update. For example, the latest servicing update for 3.1 removes the previous 3.1 update upon successful installation.

Feature bands (SDK only)

Versioning for the .NET SDK works slightly differently from the .NET runtime. To align with new Visual Studio releases, .NET SDK updates sometimes include new features or new versions of components like MSBuild and NuGet. These new features or components may be incompatible with the versions that shipped in previous SDK updates for the same major or minor version.

To differentiate such updates, the .NET SDK uses the concept of feature bands. For example, the first .NET 5 SDK was 5.0.100. This release corresponds to the 5.0.1xx *feature band*. Feature bands are defined in the hundreds groups in the third section of the version number. For example, 5.0.101 and 5.0.201 are versions in two different feature bands while 5.0.101 and 5.0.199 are in the same feature band. When .NET SDK 5.0.101 is installed, .NET SDK 5.1.100 is removed from the machine if it exists. When .NET SDK 5.0.200 is installed on the same machine, .NET SDK 5.0.101 isn't removed.

For more information about the relationship between .NET SDK and Visual Studio versions, see [.NET SDK, MSBuild, and Visual Studio versioning](#).

Runtime roll forward and compatibility

Major and minor updates install side by side with previous versions. An application built to target a specific *major.minor* version continues to use that targeted runtime even if a newer version is installed. The app doesn't automatically roll forward to use a newer *major.minor* version of the runtime unless you opt in for this behavior. An application that was built to target .NET Core 3.0 doesn't automatically start running on .NET Core 3.1. We recommend rebuilding the app and testing against a newer major or minor runtime version before deploying to production. For more information, see [Framework-dependent apps roll forward](#) and [Self-contained deployment runtime roll forward](#).

Servicing updates are treated differently from major and minor releases. An application built to target .NET 7 runs on the 7.0.0 runtime by default. It automatically rolls forward

to use a newer 7.0.1 runtime when that servicing update is installed. This behavior is the default because we want security fixes to be used as soon as they're installed without any other action needed. You can opt out from this default roll forward behavior.

.NET version lifecycles

.NET versions adopt the [modern lifecycle](#) rather than the [fixed lifecycle](#) that was used for .NET Framework releases. Products that adopt a modern lifecycle have a service-like support model, with shorter support periods and more frequent releases.

Release tracks

There are two support tracks for releases:

- *Standard Term Support (STS) releases*

These versions are supported until 6 months after the next major or minor release ships.

Example:

- .NET 5 is an STS release and was released in November 2020. It was supported for 18 months, until May 2022.
- .NET 7 is an STS release and was released in November 2022. It's supported for 18 months, until May 2024.

- *Long Term Support (LTS) releases*

These versions are supported for a minimum of 3 years, or 1 year after the next LTS release ships if that date is later.

Example:

- .NET Core 3.1 is an LTS release and was released in December 2019. It was supported for 3 years, until December 2022.
- .NET 6 is an LTS release and was released in November, 2021. It's supported for 3 years, until November 2024.

Releases alternate between LTS and STS, so it's possible for an earlier release to be supported longer than a later release. For example, .NET Core 3.1 was an LTS release with support through December 2022. The .NET 5 release shipped almost a year later but went out of support earlier, in May 2022.

Servicing updates ship monthly and include both security and non-security (reliability, compatibility, and stability) fixes. Servicing updates are supported until the next

servicing update is released. Servicing updates have runtime roll forward behavior. That means that applications default to running on the latest installed runtime servicing update.

How to choose a release

If you're building a service and expect to continue updating it on a regular basis, then an STS release like the .NET 7 runtime may be your best option to stay up to date with the latest features .NET has to offer.

If you're building a client application that will be distributed to consumers, stability might be more important than access to the latest features. Your application might need to be supported for a certain period before the consumer can upgrade to the next version of the application. In that case, an LTS release like the .NET 6 runtime could be the right option.

 **Note**

We recommend upgrading to the latest SDK version, even if it's an STS release, as it can target all available runtimes.

Support for servicing updates

.NET servicing updates are supported until the next servicing update is released. The release cadence is monthly.

You need to regularly install servicing updates to ensure that your apps are in a secure and supported state. For example, if the latest servicing update for .NET 7 is 7.0.8 and we ship 7.0.9, then 7.0.8 is no longer the latest. The supported servicing level for .NET 7 is then 7.0.9.

For information about the latest servicing updates for each major and minor version, see the [.NET downloads page](#).

End of support

End of support refers to the date after which Microsoft no longer provides fixes, updates, or technical assistance for a product version. Before this date, make sure you have moved to using a supported version. Versions that are out of support no longer

receive security updates that protect your applications and data. For the supported date ranges for each version of .NET, see the [Support Policy](#).

Supported operating systems

.NET can be run on a range of operating systems. Each of these operating systems has a lifecycle defined by its sponsor organization (for example, Microsoft, Red Hat, or Apple). We take these lifecycle schedules into account when adding and removing support for operating system versions.

When an operating system version goes out of support, we stop testing that version and providing support for that version. Users need to move forward to a supported operating system version to get support.

For more information, see the [.NET OS Lifecycle Policy](#).

Get support

You have a choice between Microsoft assisted support and Community support.

Microsoft support

For assisted support, [contact a Microsoft Support Professional](#).

You need to be on a supported servicing level (the latest available servicing update) to be eligible for support. If a system is running .NET 7 and the 7.0.8 servicing update has been released, then 7.0.8 needs to be installed as a first step.

Community support

For community support, see the [Community page](#).

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For

 .NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 Open a documentation issue

 Provide product feedback

more information, see [our contributor guide](#).

Ecma standards

Article • 09/17/2022

The C# Language and the Common Language Infrastructure (CLI) specifications are standardized through [Ecma International®](#). The first editions of these standards were published by Ecma in December 2001.

Subsequent revisions to the standards have been developed by the TC49-TG2 (C#) and TC49-TG3 (CLI) task groups within the Programming Languages Technical Committee ([TC49](#)), and adopted by the Ecma General Assembly and subsequently by ISO/IEC JTC 1 via the ISO Fast-Track process.

Latest standards

The following official Ecma documents are available for [C#](#) and the [CLI](#) ([TR-84](#)):

- The C# Language Standard (version 7): [ECMA-334.pdf](#)
- The Common Language Infrastructure: [ECMA-335.pdf](#).
- Information Derived from the Partition IV XML File: [ECMA TR/84](#) format.

The official ISO/IEC documents are available from the ISO/IEC [Publicly Available Standards](#) page. These links are direct from that page:

- Information technology - Programming languages - C#: [ISO/IEC 23270:2018](#)
- Information technology — Common Language Infrastructure (CLI) Partitions I to VI: [ISO/IEC 23271:2012](#)
- Information technology — Common Language Infrastructure (CLI) — Technical Report on Information Derived from Partition IV XML File: [ISO/IEC TR 23272:2011](#)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



.NET feedback

.NET is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

.NET glossary

Article • 05/04/2023

The primary goal of this glossary is to clarify meanings of selected terms and acronyms that appear frequently in the .NET documentation.

AOT

Ahead-of-time compiler.

Similar to [JIT](#), this compiler also translates [IL](#) to machine code. In contrast to JIT compilation, AOT compilation happens before the application is executed and is usually performed on a different machine. Because AOT tool chains don't compile at run time, they don't have to minimize time spent compiling. That means they can spend more time optimizing. Since the context of AOT is the entire application, the AOT compiler also performs cross-module linking and whole-program analysis, which means that all references are followed and a single executable is produced.

See [CoreRT](#) and [.NET Native](#).

app model

A [workload-specific API](#). Here are some examples:

- ASP.NET
- ASP.NET Web API
- Entity Framework (EF)
- Windows Presentation Foundation (WPF)
- Windows Communication Foundation (WCF)
- Windows Workflow Foundation (WF)
- Windows Forms (WinForms)

ASP.NET

The original ASP.NET implementation that ships with the .NET Framework, also known as ASP.NET 4.x.

Sometimes ASP.NET is an umbrella term that refers to both the original ASP.NET and ASP.NET Core. The meaning that the term carries in any given instance is determined by

context. Refer to ASP.NET 4.x when you want to make it clear that you're not using ASP.NET to mean both implementations.

See [ASP.NET documentation](#).

ASP.NET Core

A cross-platform, high-performance, open-source implementation of ASP.NET.

See [ASP.NET Core documentation](#).

assembly

A *.dll* or *.exe* file that can contain a collection of APIs that can be called by applications or other assemblies.

An assembly may include types such as interfaces, classes, structures, enumerations, and delegates. Assemblies in a project's *bin* folder are sometimes referred to as *binaries*. See also [library](#).

BCL

Base Class Library.

A set of libraries that comprise the System.* (and to a limited extent Microsoft.*) namespaces. The BCL is a general purpose, lower-level framework that higher-level application frameworks, such as ASP.NET Core, build on.

The source code of the BCL for [.NET 5 \(and .NET Core\) and later versions](#) is contained in the [.NET runtime repository](#). Most of these BCL APIs are also available in .NET Framework, so you can think of this source code as a fork of the .NET Framework BCL source code.

The following terms often refer to the same collection of APIs that BCL refers to:

- [core .NET libraries](#)
- [framework libraries](#)
- [runtime libraries](#)
- [shared framework](#)

CLR

Common Language Runtime.

The exact meaning depends on the context. Common Language Runtime usually refers to the runtime of [.NET Framework](#) or the runtime of [.NET 5 \(and .NET Core\) and later versions](#).

A CLR handles memory allocation and management. A CLR is also a virtual machine that not only executes apps but also generates and compiles code on-the-fly using a [JIT](#) compiler.

The CLR implementation for .NET Framework is Windows only.

The CLR implementation for .NET 5 and later versions (also known as the Core CLR) is built from the same code base as the .NET Framework CLR. Originally, the Core CLR was the runtime of Silverlight and was designed to run on multiple platforms, specifically Windows and OS X. It's still a [cross-platform](#) runtime, now including support for many Linux distributions.

See also [runtime](#).

Core CLR

The Common Language Runtime for [.NET 5 \(and .NET Core\) and later versions](#).

See [CLR](#).

CoreRT

In contrast to the [CLR](#), CoreRT is not a virtual machine, which means it doesn't include the facilities to generate and run code on-the-fly because it doesn't include a [JIT](#). It does, however, include the [GC](#) and the ability for run-time type identification (RTTI) and reflection. However, its type system is designed so that metadata for reflection isn't required. Not requiring metadata enables having an [AOT](#) tool chain that can link away superfluous metadata and (more importantly) identify code that the app doesn't use. CoreRT is in development.

See [Intro to CoreRT](#) and [.NET Runtime Lab](#).

cross-platform

The ability to develop and execute an application that can be used on multiple different operating systems, such as Linux, Windows, and iOS, without having to rewrite

specifically for each one. This enables code reuse and consistency between applications on different platforms.

See [platform](#).

ecosystem

All of the runtime software, development tools, and community resources that are used to build and run applications for a given technology.

The term ".NET ecosystem" differs from similar terms such as ".NET stack" in its inclusion of third-party apps and libraries. Here's an example in a sentence:

- "The motivation behind [.NET Standard](#) was to establish greater uniformity in the .NET ecosystem."

framework

In general, a comprehensive collection of APIs that facilitates development and deployment of applications that are based on a particular technology. In this general sense, ASP.NET Core and Windows Forms are examples of application frameworks. The words **framework** and [library](#) are often used synonymously.

The word "framework" has a different meaning in the following terms:

- [framework libraries](#)
- [.NET Framework](#)
- [shared framework](#)
- [target framework](#)
- [TFM \(target framework moniker\)](#)
- [framework-dependent app](#)

Sometimes "framework" refers to an [implementation of .NET](#). For example, an article may call .NET 5+ a framework.

framework libraries

Meaning depends on context. May refer to the framework libraries for [.NET 5 \(and .NET Core\) and later versions](#), in which case it refers to the same libraries that [BCL](#) refers to. It may also refer to the [ASP.NET Core](#) framework libraries, which build on the BCL and provide additional APIs for web apps.

GC

Garbage collector.

The garbage collector is an implementation of automatic memory management. The GC frees memory occupied by objects that are no longer in use.

See [Garbage Collection](#).

IL

Intermediate language.

Higher-level .NET languages, such as C#, compile down to a hardware-agnostic instruction set, which is called Intermediate Language (IL). IL is sometimes referred to as MSIL (Microsoft IL) or CIL (Common IL).

JIT

Just-in-time compiler.

Similar to [AOT](#), this compiler translates [IL](#) to machine code that the processor understands. Unlike AOT, JIT compilation happens on demand and is performed on the same machine that the code needs to run on. Since JIT compilation occurs during execution of the application, the compile time is part of the run time. Thus, JIT compilers have to balance time spent optimizing code against the savings that the resulting code can produce. But a JIT knows the actual hardware and can free developers from having to ship different implementations.

implementation of .NET

An implementation of .NET includes:

- One or more runtimes. Examples: [CLR](#), [CoreRT](#).
- A class library that implements a version of .NET Standard and may include additional APIs. Examples: the [BCLs](#) for [.NET Framework](#) and [.NET 5 \(and .NET Core and later versions\)](#).
- Optionally, one or more application frameworks. Examples: [ASP.NET](#), Windows Forms, and WPF are included in .NET Framework and .NET 5+.
- Optionally, development tools. Some development tools are shared among multiple implementations.

Examples of .NET implementations:

- [.NET Framework](#)
- [.NET 5 \(and .NET Core\) and later versions](#)
- [Universal Windows Platform \(UWP\)](#)
- [Mono](#)

For more information, see [.NET implementations](#).

library

A collection of APIs that can be called by apps or other libraries. A .NET library is composed of one or more [assemblies](#).

The words **library** and **framework** are often used synonymously.

Mono

An open source, [cross-platform .NET implementation](#) that's mainly used when a small runtime is required. It is the runtime that powers Xamarin applications on Android, Mac, iOS, tvOS, and watchOS and is focused primarily on apps that require a small footprint.

It supports all of the currently published .NET Standard versions.

Historically, Mono implemented the larger API of the .NET Framework and emulated some of the most popular capabilities on Unix. It is sometimes used to run .NET applications that rely on those capabilities on Unix.

Mono is typically used with a [just-in-time compiler](#), but it also features a full [static compiler \(ahead-of-time compilation\)](#) that is used on platforms like iOS.

For more information, see the [Mono documentation](#).

Native AOT

A deployment mode where the app is self-contained and has been [ahead-of-time](#) compiled to native code at the time of publish. Native AOT apps don't use a [JIT](#) compiler at run time. They can run on machines that don't have the .NET runtime installed.

For more information, see [Native AOT deployment](#).

.NET

- In general, .NET is the umbrella term for [.NET Standard](#) and all [.NET implementations](#) and workloads.
- More specifically, .NET refers to the implementation of .NET that is recommended for all new development: [.NET 5 \(and .NET Core\) and later versions](#).

For example, the first meaning is intended in phrases such as "implementations of .NET" or "the .NET development platform." The second meaning is intended in names such as [.NET SDK](#) and [.NET CLI](#).

.NET is always fully capitalized, never ".Net".

See [.NET documentation](#)

.NET 5+

The plus sign after a version number means "and later versions." See [.NET 5 and later versions](#).

.NET 5 and later versions

A cross-platform, high-performance, open-source implementation of .NET. Also referred to as .NET 5+. Includes a Common Language Runtime ([CLR](#)), an [AOT](#) runtime ([CoreRT](#), in development), a Base Class Library ([BCL](#)), and the [.NET SDK](#).

Earlier versions of this .NET implementation are known as [.NET Core](#). .NET 5 is the next version following .NET Core 3.1. Version 4 was skipped to avoid confusing this newer implementation of .NET with the older implementation that is known as [.NET Framework](#). The current version of .NET Framework is 4.8.

See [.NET documentation](#).

.NET CLI

A cross-platform toolchain for developing applications and libraries for [.NET 5 \(and .NET Core\) and later versions](#). Also known as the .NET Core CLI.

See [.NET CLI](#).

.NET Core

See [.NET 5 and later versions](#).

.NET Framework

An [implementation of .NET](#) that runs only on Windows. Includes the Common Language Runtime ([CLR](#)), the Base Class Library ([BCL](#)), and application framework libraries such as [ASP.NET](#), Windows Forms, and WPF.

See [.NET Framework Guide](#).

.NET Native

A compiler tool chain that produces native code ahead-of-time ([AOT](#)), as opposed to just-in-time ([JIT](#)).

Compilation happens on the developer's machine similar to the way a C++ compiler and linker works. It removes unused code and spends more time optimizing it. It extracts code from libraries and merges them into the executable. The result is a single module that represents the entire app.

UWP is the application framework supported by .NET Native.

See [.NET Native documentation](#).

.NET SDK

A set of libraries and tools that allow developers to create .NET applications and libraries for [.NET 5 \(and .NET Core\) and later versions](#). Also known as the .NET Core SDK.

Includes the [.NET CLI](#) for building apps, .NET libraries and runtime for building and running apps, and the dotnet executable (`dotnet.exe`) that runs CLI commands and runs applications.

See [.NET SDK Overview](#).

.NET Standard

A formal specification of .NET APIs that are available in each [.NET implementation](#).

The .NET Standard specification is sometimes called a library. Because a library includes API implementations, not only specifications (interfaces), it's misleading to call .NET Standard a "library."

See [.NET Standard](#).

NGen

Native (image) generation.

You can think of this technology as a persistent [JIT](#) compiler. It usually compiles code on the machine where the code is executed, but compilation typically occurs at install time.

package

A NuGet package—or just a package—is a *.zip* file with one or more assemblies of the same name along with additional metadata such as the author name.

The *.zip* file has a *.nupkg* extension and may contain assets, such as *.dll* files and *.xml* files, for use with multiple [target frameworks](#) and versions. When installed in an app or library, the appropriate assets are selected based on the target framework specified by the app or library. The assets that define the interface are in the *ref* folder, and the assets that define the implementation are in the *lib* folder.

platform

An operating system and the hardware it runs on, such as Windows, macOS, Linux, iOS, and Android.

Here are examples of usage in sentences:

- ".NET Core is a cross-platform implementation of .NET."
- "PCL profiles represent Microsoft platforms, while .NET Standard is agnostic to platform."

Legacy .NET documentation sometimes uses ".NET platform" to mean either an [implementation of .NET](#) or the [.NET stack](#) including all implementations. Both of these usages tend to get confused with the primary (OS/hardware) meaning, so we try to avoid these usages.

"Platform" has a different meaning in the phrase "developer platform," which refers to software that provides tools and libraries for building and running apps. .NET is a cross-platform, open-source developer platform for building many different types of applications.

POCO

A POCO—or a plain old class/[CLR](#) object—is a .NET data structure that contains only public properties or fields. A POCO shouldn't contain any other members, such as:

- methods
- events
- delegates

These objects are used primarily as data transfer objects (DTOs). A pure POCO will not inherit another object, or implement an interface. It's common for POCOs to be used with serialization.

runtime

In general, the execution environment for a managed program. The OS is part of the runtime environment but is not part of the .NET runtime. Here are some examples of .NET runtimes in this sense of the word:

- Common Language Runtime ([CLR](#))
- .NET Native (for UWP)
- Mono runtime

The word "runtime" has a different meaning in some contexts:

- *.NET runtime* on the [.NET 5 download page](#).

You can download the *.NET runtime* or other runtimes, such as the *ASP.NET Core runtime*. A *runtime* in this usage is the set of components that must be installed on a machine to run a [framework-dependent](#) app on the machine. The .NET runtime includes the [CLR](#) and the [.NET shared framework](#), which provides the [BCL](#).

- *.NET runtime libraries*

Refers to the same libraries that [BCL](#) refers to. However, other runtimes, such as the ASP.NET Core runtime, have different [shared frameworks](#), with additional libraries that build on the BCL.

- [Runtime Identifier \(RID\)](#).

Runtime here means the OS platform and CPU architecture that a .NET app runs on, for example: `linux-x64`.

- Sometimes "runtime" is used in the sense of an [implementation of .NET](#), as in the following examples:

- "The various .NET runtimes implement specific versions of .NET Standard. ... Each .NET runtime version advertises the highest .NET Standard version it supports ..."
- "Libraries that are intended to run on multiple runtimes should target this framework." (referring to .NET Standard)

shared framework

Meaning depends on context. The *.NET shared framework* refers to the libraries included in the [.NET runtime](#). In this case, the *shared framework* for [.NET 5](#) (and [.NET Core](#)) and [later versions](#) refers to the same libraries that [BCL](#) refers to.

There are other shared frameworks. The *ASP.NET Core shared framework* refers to the libraries included in the [ASP.NET Core runtime](#), which includes the BCL plus additional APIs for use by web apps.

For [framework-dependent apps](#), the shared framework consists of libraries that are contained in assemblies installed in a folder on the machine that runs the app. For [self-contained apps](#), the shared framework assemblies are included with the app.

For more information, see [Deep-dive into .NET Core primitives, part 2: the shared framework](#).

stack

A set of programming technologies that are used together to build and run applications.

"The .NET stack" refers to .NET Standard and all .NET implementations. The phrase "a .NET stack" may refer to one implementation of .NET.

target framework

The collection of APIs that a .NET app or library relies on.

An app or library can target a version of [.NET Standard](#) (for example, .NET Standard 2.0), which is a specification for a standardized set of APIs across all [.NET implementations](#). An app or library can also target a version of a specific .NET implementation, in which case it gets access to implementation-specific APIs. For example, an app that targets Xamarin.iOS gets access to Xamarin-provided iOS API wrappers.

For some target frameworks (for example, [.NET Framework](#)) the available APIs are defined by the assemblies that a .NET implementation installs on a system, which may

include application framework APIs (for example, ASP.NET, WinForms). For package-based target frameworks, the framework APIs are defined by the packages installed in the app or library.

See [Target Frameworks](#).

TFM

Target framework moniker.

A standardized token format for specifying the [target framework](#) of a .NET app or library. Target frameworks are typically referenced by a short name, such as `net462`. Long-form TFMs (such as `.NETFramework,Version=4.6.2`) exist but are not generally used to specify a target framework.

See [Target Frameworks](#).

UWP

Universal Windows Platform.

An [implementation of .NET](#) that is used for building touch-enabled Windows applications and software for the Internet of Things (IoT). It's designed to unify the different types of devices that you may want to target, including PCs, tablets, phones, and even the Xbox. UWP provides many services, such as a centralized app store, an execution environment (AppContainer), and a set of Windows APIs to use instead of Win32 (WinRT). Apps can be written in C++, C#, Visual Basic, and JavaScript. When using C# and Visual Basic, the .NET APIs are provided by [.NET 5 \(and .NET Core\) and later versions](#).

workload

A type of app someone is building. More generic than [app model](#). For example, at the top of every .NET documentation page, including this one, is a drop-down list for **Workloads**, which lets you switch to documentation for **Web, Mobile, Cloud, Desktop, and Machine Learning & Data**.

In some contexts, *workload* refers to a collection of Visual Studio features that you can choose to install to support a particular type of app. For an example, see [Select a workload](#).

See also

- [.NET fundamentals](#)
- [.NET Framework Guide](#)
- [ASP.NET Overview](#)
- [ASP.NET Core Overview](#)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

C# console app template generates top-level statements

Article • 02/16/2023

Starting with .NET 6, the project template for new C# console apps generates the following code in the *Program.cs* file:

C#

```
// See https://aka.ms/new-console-template for more information
Console.WriteLine("Hello, World!");
```

The new output uses recent C# features that simplify the code you need to write for a program. For .NET 5 and earlier versions, the console app template generates the following code:

C#

```
using System;

namespace MyApp // Note: actual namespace depends on the project name.
{
    internal class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello World!");
        }
    }
}
```

These two forms represent the same program. Both are valid with C# 10.0. When you use the newer version, you only need to write the body of the `Main` method. The compiler generates a `Program` class with an entry point method and places all your top level statements in that method. The name of the generated method isn't `Main`, it's an implementation detail that your code can't reference directly. You don't need to include the other program elements, the compiler generates them for you. You can learn more about the code the compiler generates when you use top level statements in the article on [top level statements](#) in the C# Guide's fundamentals section.

You have two options to work with tutorials that haven't been updated to use .NET 6+ templates:

- Use the new program style, adding new top-level statements as you add features.
- Convert the new program style to the older style, with a `Program` class and a `Main` method.

If you want to use the old templates, see [Use the old program style](#) later in this article.

Use the new program style

The features that make the new program simpler are *top-level statements*, *global using directives*, and *implicit using directives*.

The term [*top-level statements*](#) means the compiler generates the class and method elements for your main program. The compiler generated class and entry point method are declared in the global namespace. You can look at the code for the new application and imagine that it contains the statements inside the `Main` method generated by earlier templates, but in the global namespace.

You can add more statements to the program, just like you can add more statements to your `Main` method in the traditional style. You can [access args \(command-line arguments\)](#), [use await](#), and [set the exit code](#). You can even add functions. They're created as local functions nested inside the generated entry point method. Local functions can't include any access modifiers (for example, `public` or `protected`).

Both top-level statements and [*implicit using directives*](#) simplify the code that makes up your application. To follow an existing tutorial, add any new statements to the `Program.cs` file generated by the template. You can imagine that the statements you write are between the open and closing braces in the `Main` method in the instructions of the tutorial.

If you'd prefer to use the older format, you can copy the code from the second example in this article, and continue the tutorial as before.

You can learn more about top-level statements in the tutorial exploration on [top-level statements](#).

Implicit using directives

The term *implicit using directives* means the compiler automatically adds a set of [*using directives*](#) based on the project type. For console applications, the following directives are implicitly included in the application:

- `using System;`

- `using System.IO;`
- `using System.Collections.Generic;`
- `using System.Linq;`
- `using System.Net.Http;`
- `using System.Threading;`
- `using System.Threading.Tasks;`

Other application types include more namespaces that are common for those application types.

If you need `using` directives that aren't implicitly included, you can add them to the .cs file that contains top-level statements or to other .cs files. For `using` directives that you need in all of the .cs files in an application, use [global using directives](#).

Disable implicit `using` directives

If you want to remove this behavior and manually control all namespaces in your project, add `<ImplicitUsings>disable</ImplicitUsings>` to your project file in the `<PropertyGroup>` element, as shown in the following example:

XML

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    ...
    <ImplicitUsings>disable</ImplicitUsings>
  </PropertyGroup>

</Project>
```

Global `using` directives

A *global using directive* imports a namespace for your whole application instead of a single file. These global directives can be added either by adding a `<Using>` item to the project file, or by adding the `global using` directive to a code file.

You can also add a `<Using>` item with a `Remove` attribute to your project file to remove a specific *implicit using directive*. For example, if the implicit `using` directives feature is turned on with `<ImplicitUsings>enable</ImplicitUsings>`, adding the following `<Using>` item removes the `System.Net.Http` namespace from those that are implicitly imported:

XML

```
<ItemGroup>
  <Using Remove="System.Net.Http" />
</ItemGroup>
```

Use the old program style

Starting with .NET SDK 6.0.300, the `console` template has a `--use-program-main` option. Use it to create a console project that doesn't use top-level statements and has a `Main` method.

.NET CLI

```
dotnet new console --use-program-main
```

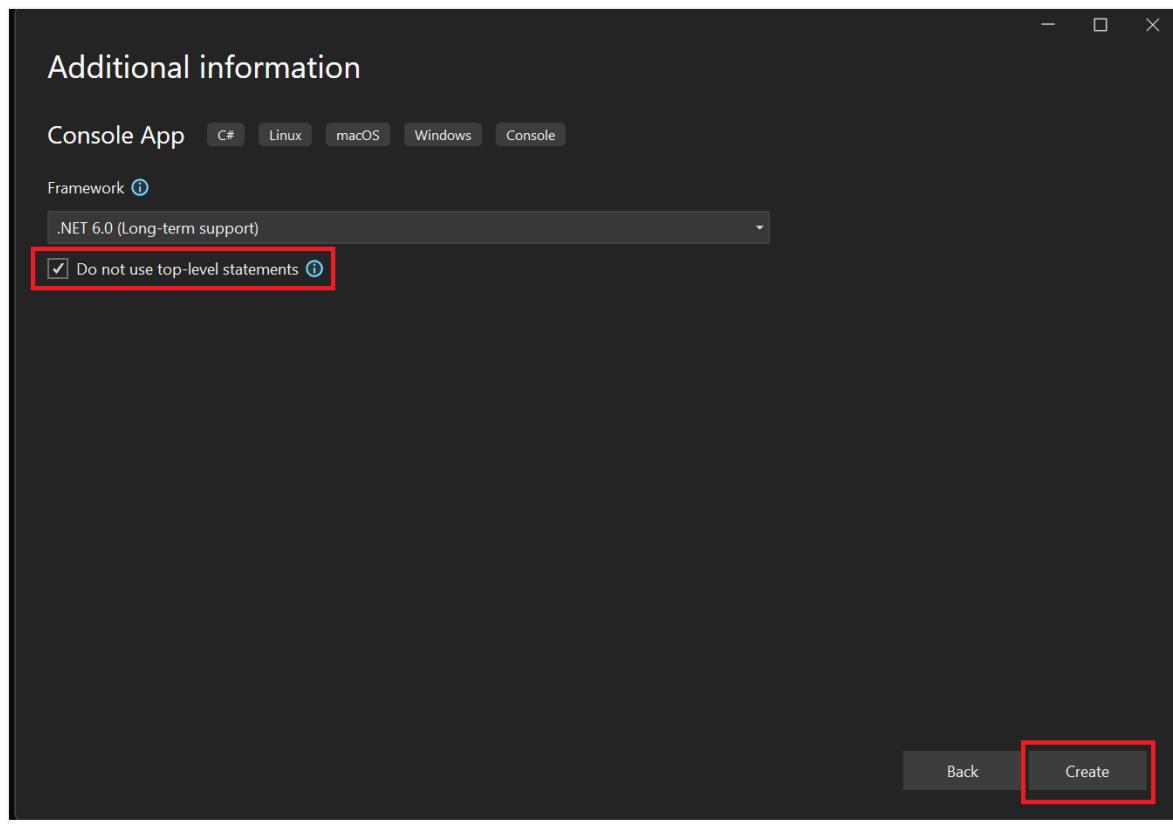
The generated `Program.cs` is as follows:

C#

```
namespace MyProject;
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("Hello, World!");
    }
}
```

Use the old program style in Visual Studio

1. When you create a new project, the setup steps will navigate to the **Additional information** setup page. On this page, select the **Do not use top-level statements** check box.



2. After your project is created, the `Program.cs` content is as follows:

```
C#  
  
namespace MyProject;  
class Program  
{  
    static void Main(string[] args)  
    {  
        Console.WriteLine("Hello, World!");  
    }  
}
```

(!) Note

Visual Studio preserves the value for the options next time you create the project based on the same template, so by default when creating Console App project next time the "Do not use top-level statements" check box will be checked. The content of the `Program.cs` file might be different to match the code style defined in the global Visual Studio text editor settings or the `EditorConfig` file.

For more information, see [Create portable, custom editor settings with EditorConfig and Options, Text Editor, C#, Advanced](#).

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

.NET is an open source project.
Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

Tutorial: Create a .NET console application using Visual Studio

Article • 08/25/2023

This tutorial shows how to create and run a .NET console application in Visual Studio 2022.

Prerequisites

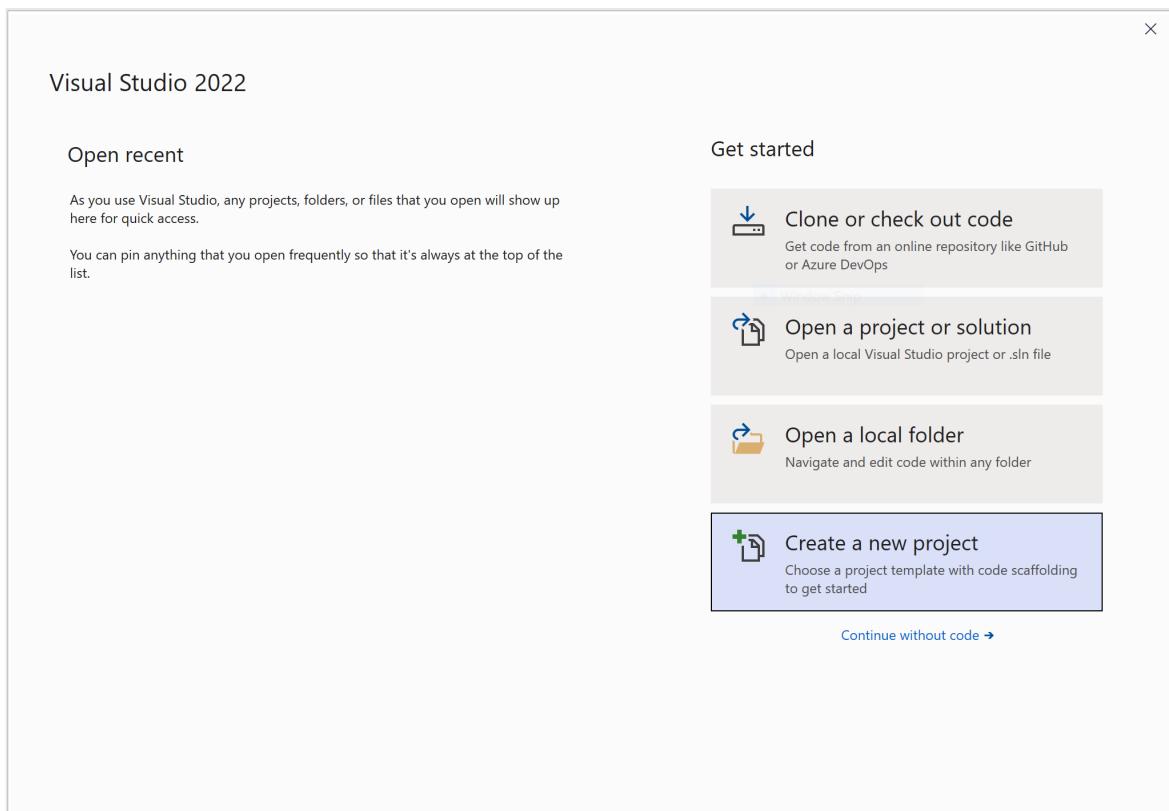
- [Visual Studio 2022 Preview](#) with the **.NET desktop development** workload installed. The .NET 8 SDK is automatically installed when you select this workload.

For more information, see [Install the .NET SDK with Visual Studio](#).

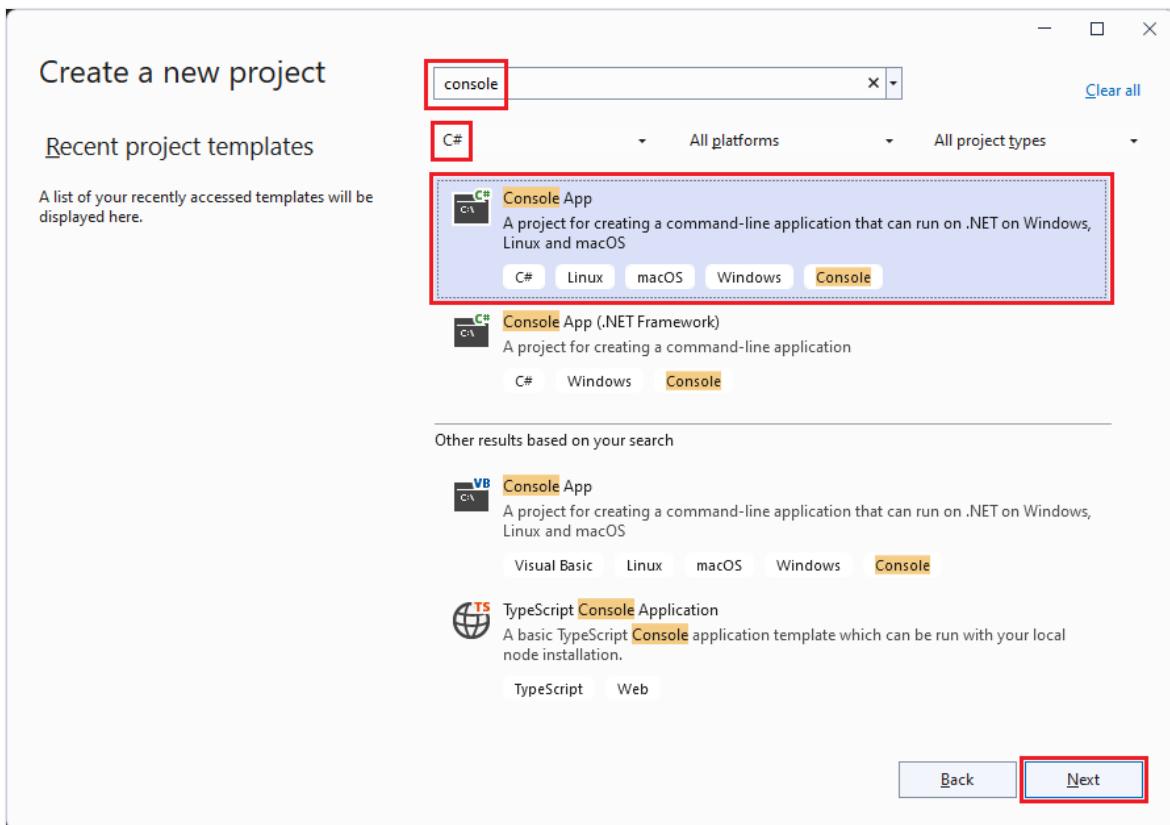
Create the app

Create a .NET console app project named "HelloWorld".

1. Start Visual Studio 2022.
2. On the start page, choose **Create a new project**.



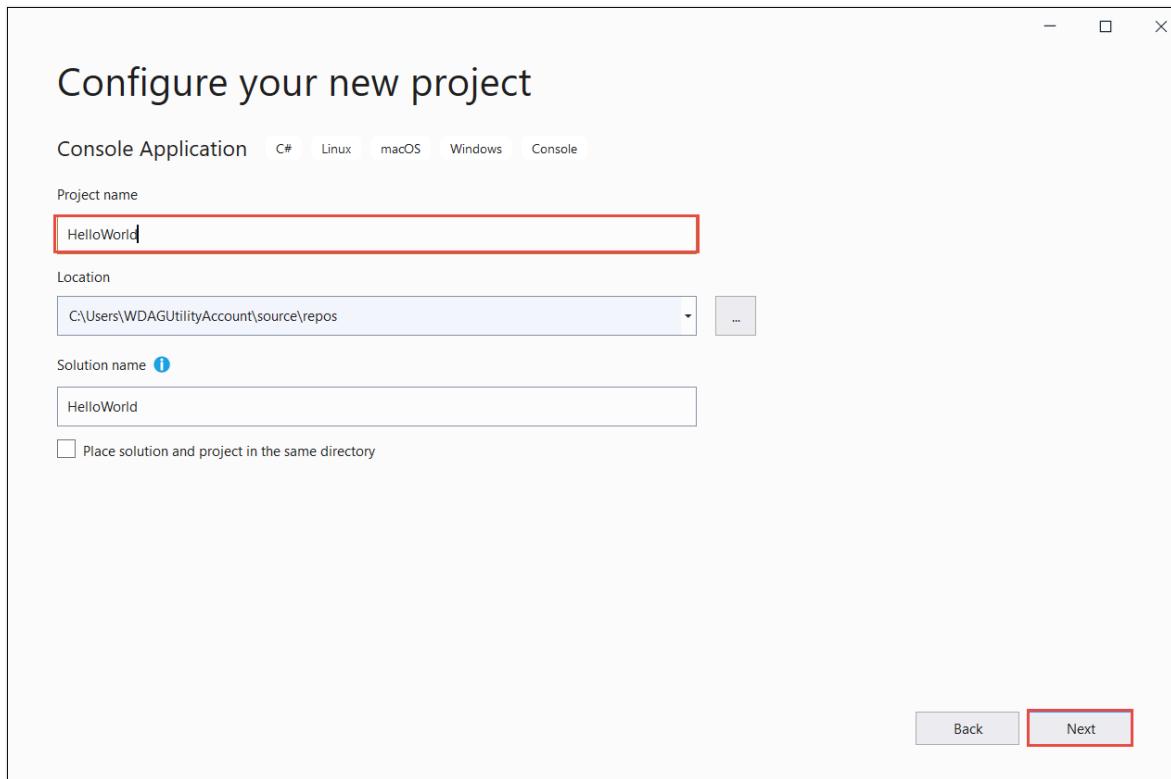
3. On the **Create a new project** page, enter **console** in the search box. Next, choose **C# or Visual Basic** from the language list, and then choose **All platforms** from the platform list. Choose the **Console App** template, and then choose **Next**.



💡 Tip

If you don't see the .NET templates, you're probably missing the required workload. Under the **Not finding what you're looking for?** message, choose the **Install more tools and features** link. The Visual Studio Installer opens. Make sure you have the **.NET desktop development** workload installed.

4. In the **Configure your new project** dialog, enter **HelloWorld** in the **Project name** box. Then choose **Next**.



5. In the Additional information dialog:

- Select .NET 8 (Preview).
- Select Do not use top-level statements.
- Select Create.

The template creates a simple application that displays "Hello, World!" in the console window. The code is in the *Program.cs* or *Program.vb* file:

```
C#  
  
namespace HelloWorld;  
  
internal class Program  
{  
    static void Main(string[] args)  
    {  
        Console.WriteLine("Hello, World!");  
    }  
}
```

If the language you want to use is not shown, change the language selector at the top of the page.

The code defines a class, `Program`, with a single method, `Main`, that takes a `String` array as an argument. `Main` is the application entry point, the method that's called automatically by the runtime when it launches the application. Any command-line

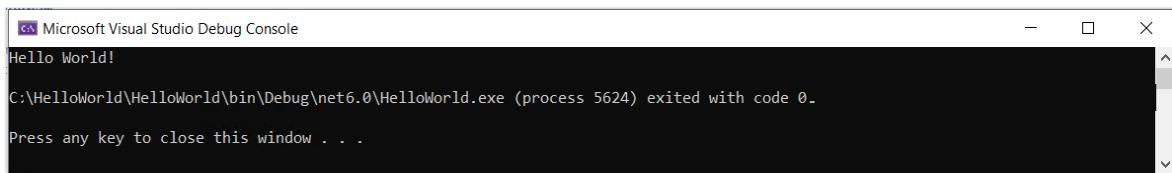
arguments supplied when the application is launched are available in the `args` array.

C# has a feature named [top-level statements](#) that lets you omit the `Program` class and the `Main` method. This tutorial doesn't use this feature. Whether you use it in your programs is a matter of style preference.

Run the app

1. Press `Ctrl + F5` to run the program without debugging.

A console window opens with the text "Hello, World!" printed on the screen. (Or "Hello World!" without a comma in the Visual Basic project template.)



2. Press any key to close the console window.

Enhance the app

Enhance the application to prompt the user for their name and display it along with the date and time.

1. In `Program.cs` or `Program.vb`, replace the contents of the `Main` method, which is the line that calls `Console.WriteLine`, with the following code:

```
C#  
  
Console.WriteLine("What is your name?");  
var name = Console.ReadLine();  
var currentDate = DateTime.Now;  
Console.WriteLine($"{Environment.NewLine}Hello, {name}, on  
{currentDate:d} at {currentDate:t}!");  
Console.Write($"{Environment.NewLine}Press any key to exit...");  
Console.ReadKey(true);
```

This code displays a prompt in the console window and waits until the user enters a string followed by the `Enter` key. It stores this string in a variable named `name`. It also retrieves the value of the `DateTime.Now` property, which contains the current local time, and assigns it to a variable named `currentDate`. And it displays these

values in the console window. Finally, it displays a prompt in the console window and calls the [Console.ReadKey\(Boolean\)](#) method to wait for user input.

[Environment.NewLine](#) is a platform-independent and language-independent way to represent a line break. Alternatives are `\n` in C# and `vbCrLf` in Visual Basic.

The dollar sign (\$) in front of a string lets you put expressions such as variable names in curly braces in the string. The expression value is inserted into the string in place of the expression. This syntax is referred to as [interpolated strings](#).

2. Press `Ctrl + F5` to run the program without debugging.
3. Respond to the prompt by entering a name and pressing the `Enter` key.

A screenshot of a .NET console application window. The text output is:
What is your name?
Maira

Hello, Maira, on 12/3/2019 at 3:36 AM!

Press any key to exit...

4. Press any key to close the console window.

Additional resources

- Standard-term support (STS) releases and long-term support (LTS) releases.

Next steps

In this tutorial, you created a .NET console application. In the next tutorial, you debug the app.

[Debug a .NET console application using Visual Studio](#)

 Collaborate with us on
GitHub

The source for this content can
be found on GitHub, where you

.NET

[.NET feedback](#)

The .NET documentation is open
source. Provide feedback here.

can also create and review issues and pull requests. For more information, see [our contributor guide](#).

 Open a documentation issue

 Provide product feedback

Tutorial: Debug a .NET console application using Visual Studio

Article • 08/25/2023

This tutorial introduces the debugging tools available in Visual Studio.

ⓘ Important

All of the keyboard shortcuts are based on the defaults from Visual Studio. Your keyboard shortcuts may vary, for more information see [Keyboard shortcuts in Visual Studio](#).

Prerequisites

- This tutorial works with the console app that you create in [Create a .NET console application using Visual Studio](#).

Use Debug build configuration

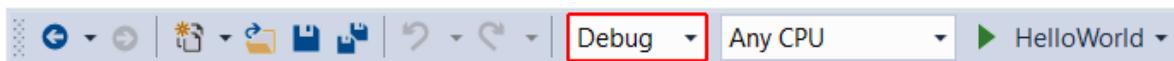
Debug and *Release* are Visual Studio's built-in build configurations. You use the *Debug* build configuration for debugging and the *Release* configuration for the final release distribution.

In the *Debug* configuration, a program compiles with full symbolic debug information and no optimization. Optimization complicates debugging, because the relationship between source code and generated instructions is more complex. The *release* configuration of a program has no symbolic debug information and is fully optimized.

By default, Visual Studio uses the *Debug* build configuration, so you don't need to change it before debugging.

1. Start Visual Studio.
2. Open the project that you created in [Create a .NET console application using Visual Studio](#).

The current build configuration is shown on the toolbar. The following toolbar image shows that Visual Studio is configured to compile the *Debug* version of the app:

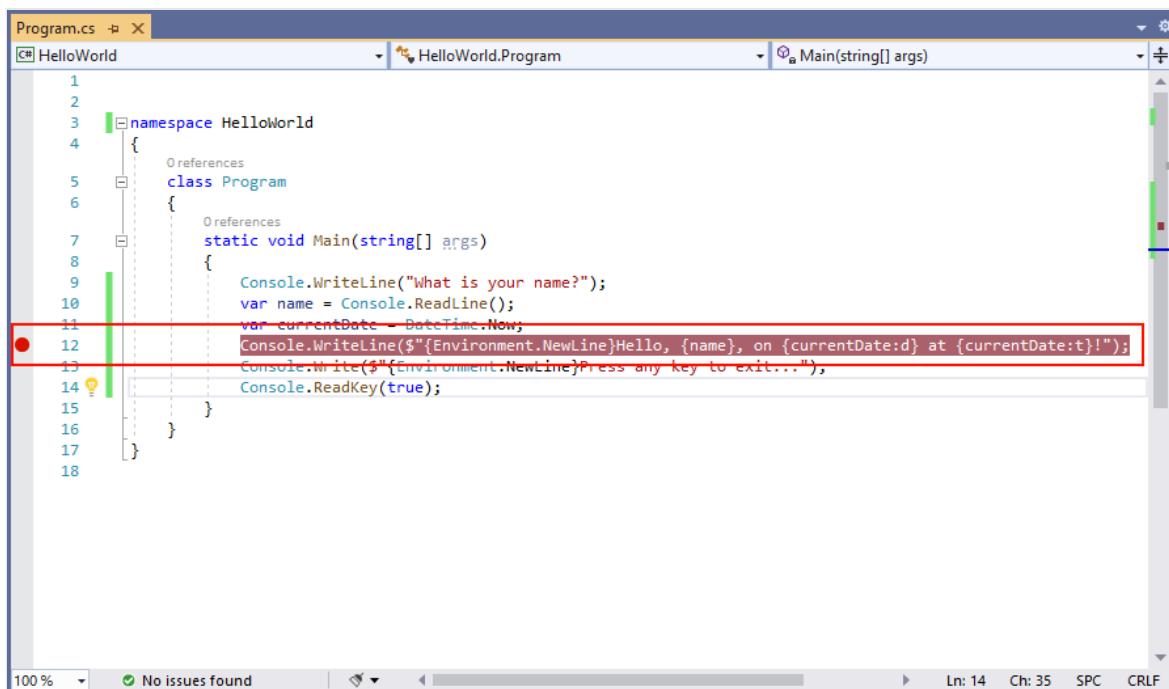


Set a breakpoint

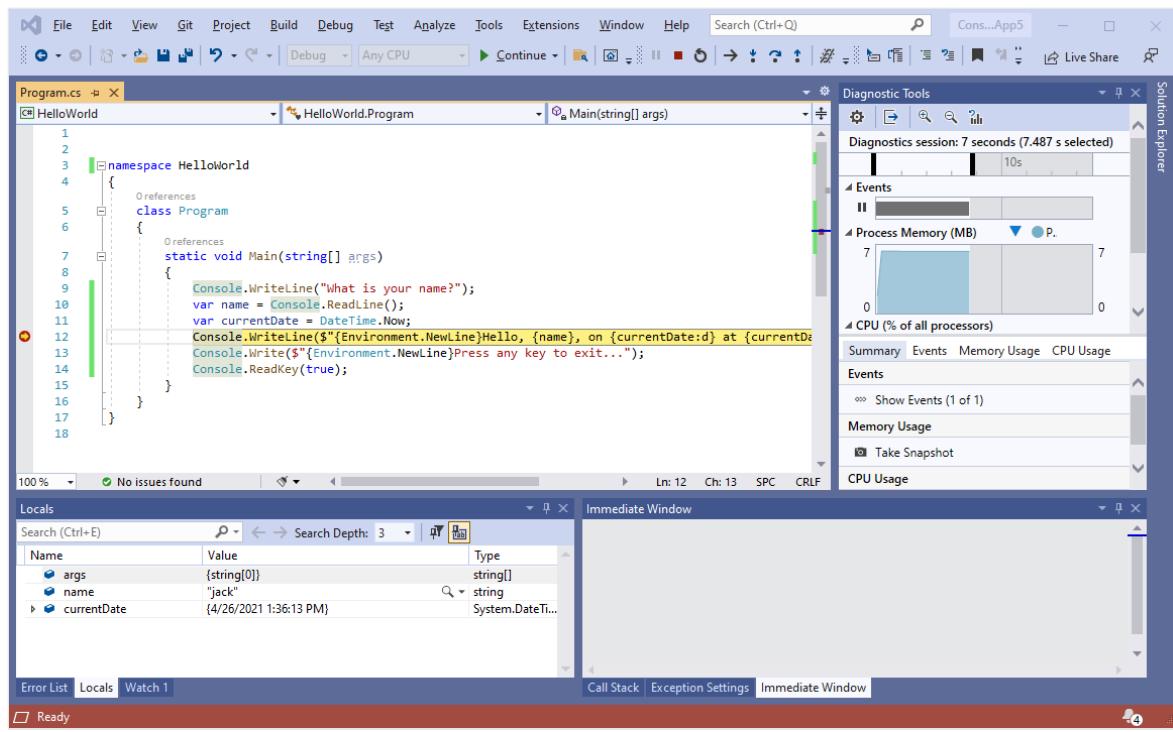
A *breakpoint* temporarily interrupts the execution of the application before the line with the breakpoint is executed.

1. Set a *breakpoint* on the line that displays the name, date, and time, by clicking in the left margin of the code window on that line. The left margin is to the left of the line numbers. Other ways to set a breakpoint are by placing the cursor in the line of code and then pressing **F9** or choosing **Debug > Toggle Breakpoint** from the menu bar.

As the following image shows, Visual Studio indicates the line on which the breakpoint is set by highlighting it and displaying a red dot in the left margin.



2. Press **F5** to run the program in Debug mode. Another way to start debugging is by choosing **Debug > Start Debugging** from the menu.
3. Enter a string in the console window when the program prompts for a name, and then press **Enter**.
4. Program execution stops when it reaches the breakpoint and before the `Console.WriteLine` method executes. The **Locals** window displays the values of variables that are defined in the currently executing method.

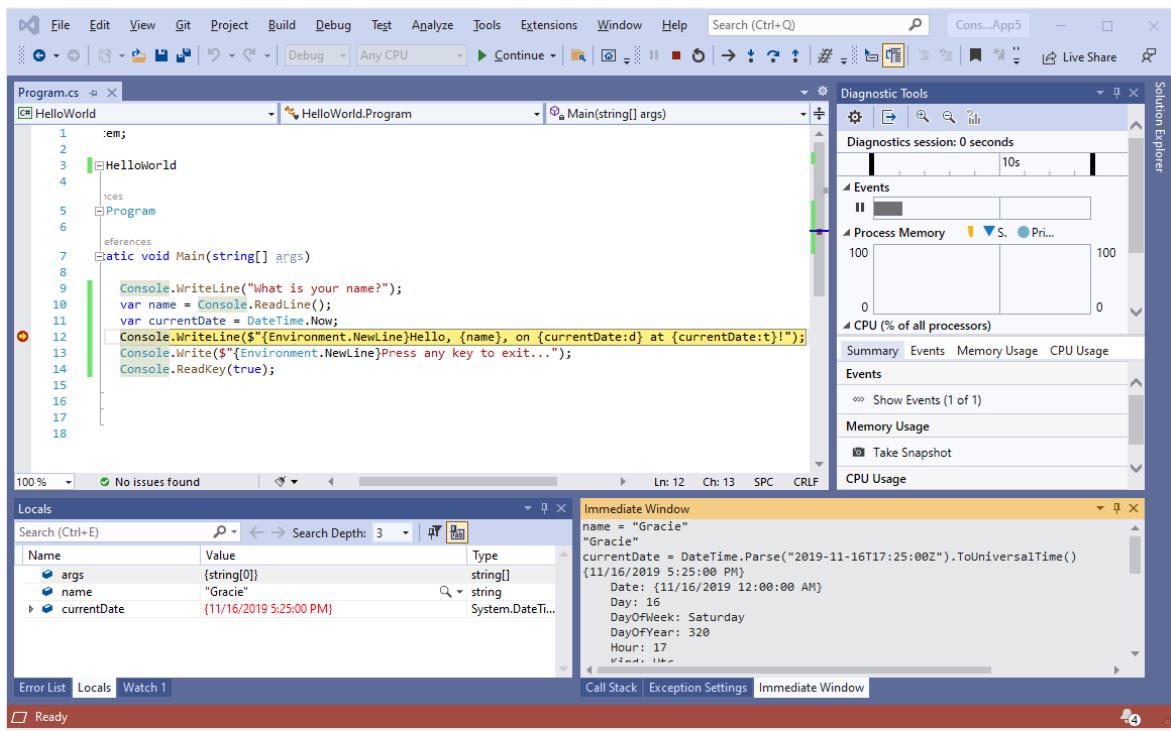


Use the Immediate window

The **Immediate** window lets you interact with the application you're debugging. You can interactively change the value of variables to see how it affects your program.

1. If the **Immediate** window is not visible, display it by choosing **Debug > Windows > Immediate**.
2. Enter `name = "Gracie"` in the **Immediate** window and press the **Enter** key.
3. Enter `currentDate = DateTime.Parse("2019-11-16T17:25:00Z").ToUniversalTime()` in the **Immediate** window and press the **Enter** key.

The **Immediate** window displays the value of the string variable and the properties of the **DateTime** value. In addition, the values of the variables are updated in the **Locals** window.



4. Press **F5** to continue program execution. Another way to continue is by choosing **Debug > Continue** from the menu.

The values displayed in the console window correspond to the changes you made in the **Immediate** window.

```

What is your name?
jack

Hello, Gracie, on 11/16/2019 at 5:25 PM!

Press any key to exit...

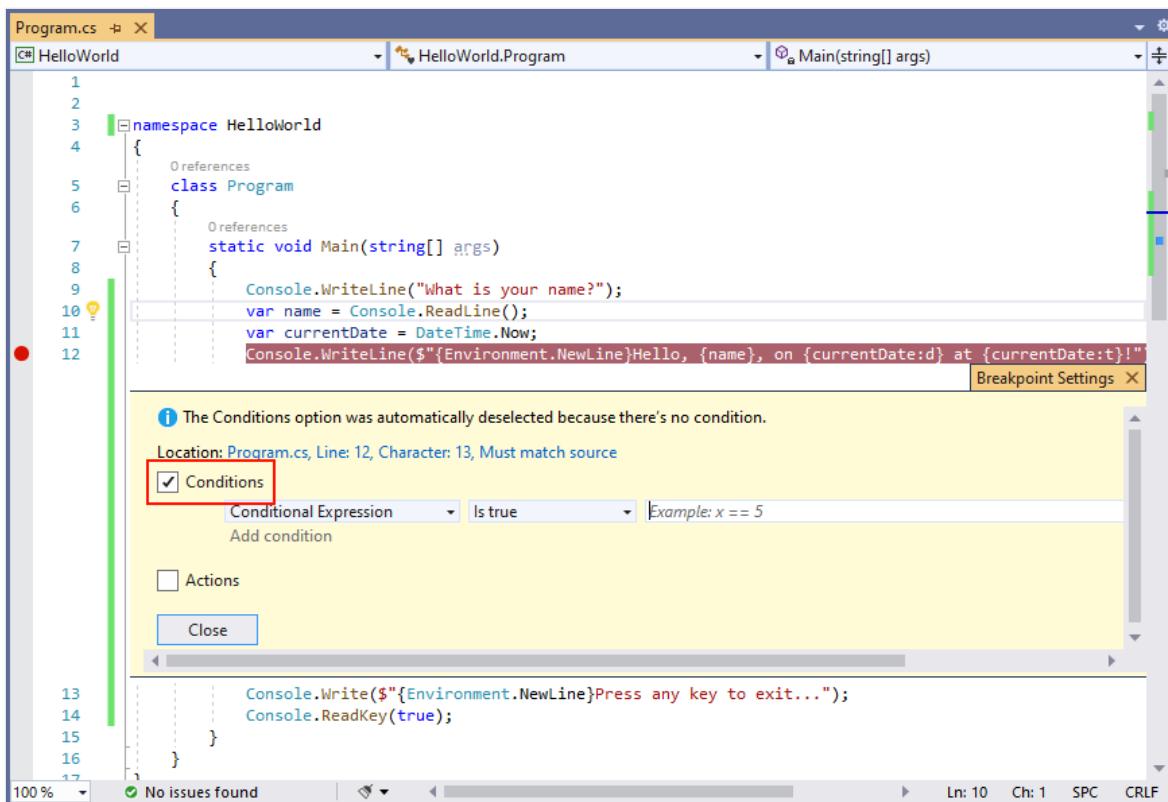
```

5. Press any key to exit the application and stop debugging.

Set a conditional breakpoint

The program displays the string that the user enters. What happens if the user doesn't enter anything? You can test this with a useful debugging feature called a *conditional breakpoint*.

1. Right-click on the red dot that represents the breakpoint. In the context menu, select **Conditions** to open the **Breakpoint Settings** dialog. Select the box for **Conditions** if it's not already selected.



2. For the **Conditional Expression**, enter the following code in the field that shows example code that tests if `x` is 5.

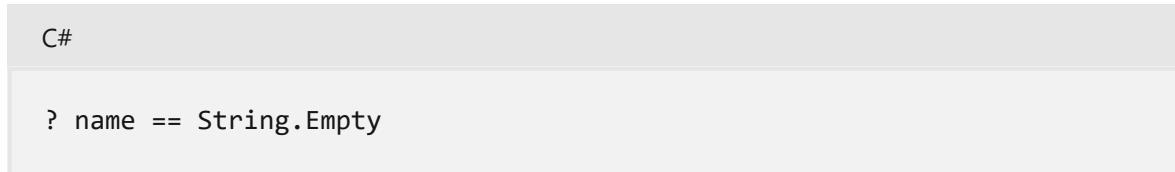
```
C#  
  
string.IsNullOrEmpty(name)
```

Each time the breakpoint is hit, the debugger calls the `String.IsNullOrEmpty(name)` method, and it breaks on this line only if the method call returns `true`.

Instead of a conditional expression, you can specify a *hit count*, which interrupts program execution before a statement is executed a specified number of times. Another option is to specify a *filter condition*, which interrupts program execution based on such attributes as a thread identifier, process name, or thread name.

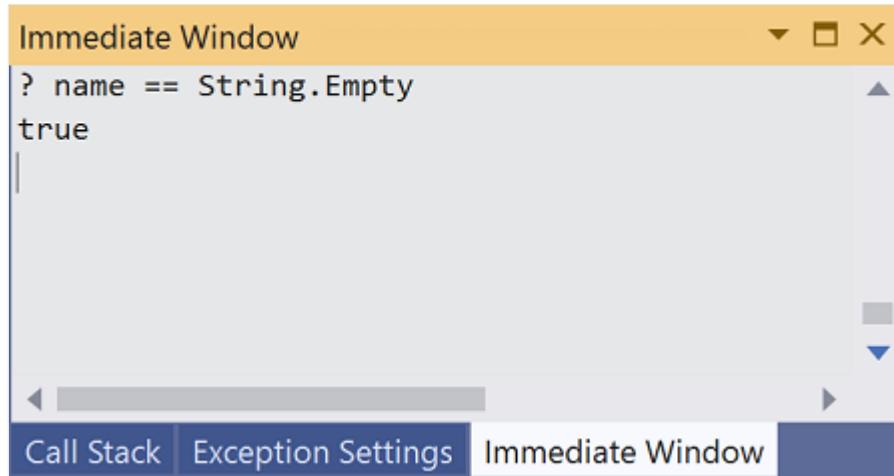
3. Select **Close** to close the dialog.
4. Start the program with debugging by pressing `F5`.
5. In the console window, press the `Enter` key when prompted to enter your name.
6. Because the condition you specified (`name` is either `null` or `String.Empty`) has been satisfied, program execution stops when it reaches the breakpoint and before the `Console.WriteLine` method executes.

7. Select the **Locals** window, which shows the values of variables that are local to the currently executing method. In this case, `Main` is the currently executing method. Observe that the value of the `name` variable is `""`, or `String.Empty`.
8. Confirm the value is an empty string by entering the following statement in the **Immediate** window and pressing `Enter`. The result is `true`.



```
C#  
? name == String.Empty
```

The question mark directs the immediate window to [evaluate an expression](#).



9. Press `F5` to continue program execution.
10. Press any key to close the console window and stop debugging.
11. Clear the breakpoint by clicking on the dot in the left margin of the code window. Other ways to clear a breakpoint are by pressing `F9` or choosing **Debug > Toggle Breakpoint** while the line of code is selected.

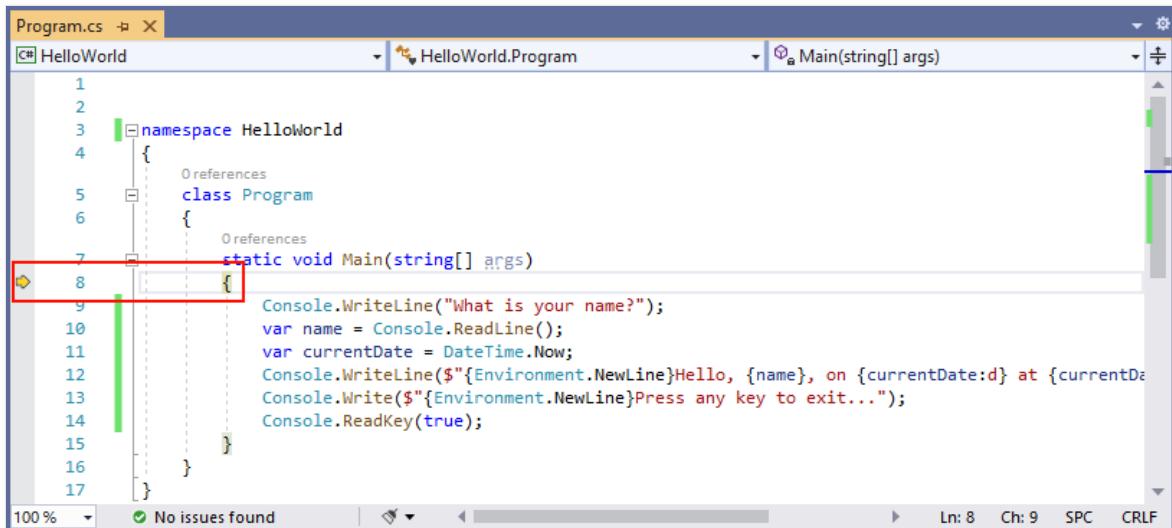
Step through a program

Visual Studio also allows you to step line by line through a program and monitor its execution. Ordinarily, you'd set a breakpoint and follow program flow through a small part of your program code. Since this program is small, you can step through the entire program.

1. Choose **Debug > Step Into**. Another way to debug one statement at a time is by pressing `F11`.

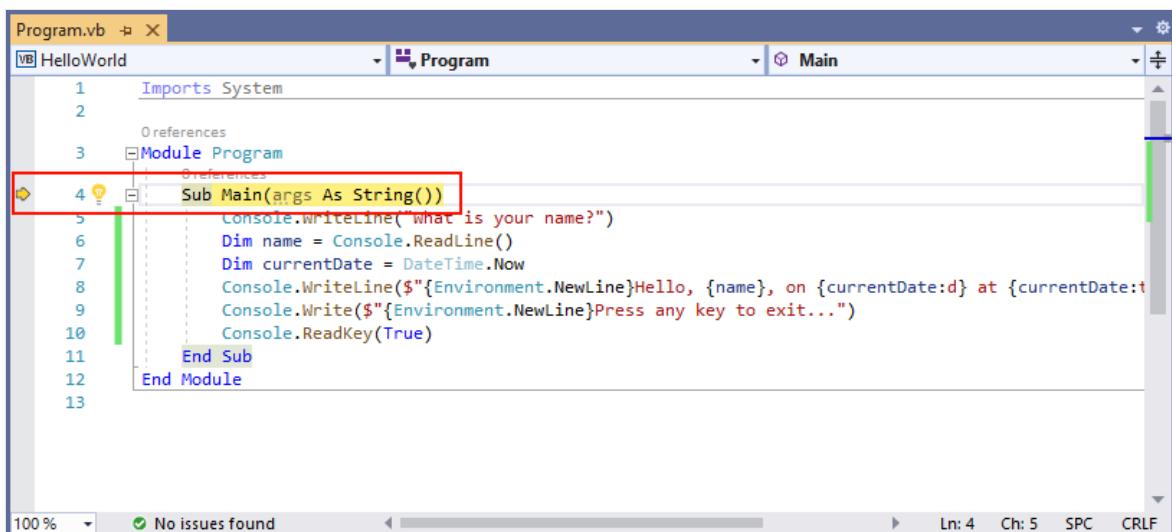
Visual Studio highlights and displays an arrow beside the next line of execution.

C#



```
1
2
3  namespace HelloWorld
4  {
5      class Program
6      {
7          static void Main(string[] args)
8          {
9              Console.WriteLine("What is your name?");
10             var name = Console.ReadLine();
11             var currentDate = DateTime.Now;
12             Console.WriteLine($"{Environment.NewLine}Hello, {name}, on {currentDate:d} at {currentDate:t}");
13             Console.Write($"{Environment.NewLine}Press any key to exit...");
14             Console.ReadKey(true);
15         }
16     }
17 }
```

Visual Basic



```
1 Imports System
2
3 Module Program
4     Sub Main(args As String())
5         Console.WriteLine("what is your name?")
6         Dim name = Console.ReadLine()
7         Dim currentDate = DateTime.Now
8         Console.WriteLine($"{Environment.NewLine}Hello, {name}, on {currentDate:d} at {currentDate:t}");
9         Console.Write($"{Environment.NewLine}Press any key to exit...")
10        Console.ReadKey(True)
11    End Sub
12 End Module
13
```

At this point, the **Locals** window shows that the `args` array is empty, and `name` and `currentDate` have default values. In addition, Visual Studio has opened a blank console window.

2. Press **F11**. Visual Studio now highlights the next line of execution. The **Locals** window is unchanged, and the console window remains blank.

C#

```
Program.cs
HelloWorld
HelloWorld.Program
Main(string[] args)

1
2
3 namespace HelloWorld
4 {
5     class Program
6     {
7         static void Main(string[] args)
8         {
9             Console.WriteLine("What is your name?"); ⏵1ms elapsed
10            var name = Console.ReadLine();
11            var currentDate = DateTime.Now;
12            Console.WriteLine($"{Environment.NewLine}Hello, {name}, on {currentDate:d} at {currentDate:t}");
13            Console.Write($"{Environment.NewLine}Press any key to exit...");
14            Console.ReadKey(true);
15        }
16    }
17 }
```

Visual Basic

```
Program.vb
VB HelloWorld
Program
Main

1 Imports System
2
3 Module Program
4     Sub Main(args As String())
5         Console.WriteLine("What is your name?") ⏵1ms elapsed
6         Dim name = Console.ReadLine()
7         Dim currentDate = DateTime.Now
8         Console.WriteLine($"{Environment.NewLine}Hello, {name}, on {currentDate:d} at {currentDate:t}");
9         Console.Write($"{Environment.NewLine}Press any key to exit...");
10        Console.ReadKey(True)
11    End Sub
12 End Module
13 
```

3. Press **F11**. Visual Studio highlights the statement that includes the `name` variable assignment. The **Locals** window shows that `name` is `null`, and the console window displays the string "What is your name?".
4. Respond to the prompt by entering a string in the console window and pressing **Enter**. The console is unresponsive, and the string you entered isn't displayed in the console window, but the `Console.ReadLine` method will nevertheless capture your input.
5. Press **F11**. Visual Studio highlights the statement that includes the `currentDate` variable assignment. The **Locals** window shows the value returned by the call to the `Console.ReadLine` method. The console window also displays the string you entered at the prompt.
6. Press **F11**. The **Locals** window shows the value of the `currentDate` variable after the assignment from the `DateTime.Now` property. The console window is unchanged.

7. Press **F11**. Visual Studio calls the `Console.WriteLine(String, Object, Object)` method.

The console window displays the formatted string.

8. Choose **Debug > Step Out**. Another way to stop step-by-step execution is by pressing **Shift + F11**.

The console window displays a message and waits for you to press a key.

9. Press any key to close the console window and stop debugging.

Use Release build configuration

Once you've tested the Debug version of your application, you should also compile and test the Release version. The Release version incorporates compiler optimizations that can sometimes negatively affect the behavior of an application. For example, compiler optimizations that are designed to improve performance can create race conditions in multithreaded applications.

To build and test the Release version of your console application, change the build configuration on the toolbar from **Debug** to **Release**.



When you press **F5** or choose **Build Solution** from the **Build** menu, Visual Studio compiles the Release version of the application. You can test it as you did the Debug version.

Next steps

In this tutorial, you used Visual Studio debugging tools. In the next tutorial, you publish a deployable version of the app.

[Publish a .NET console application using Visual Studio](#)

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For

.NET

 .NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

more information, see [our contributor guide](#).

 [Provide product feedback](#)

Tutorial: Publish a .NET console application using Visual Studio

Article • 08/25/2023

This tutorial shows how to publish a console app so that other users can run it. Publishing creates the set of files that are needed to run your application. To deploy the files, copy them to the target machine.

Prerequisites

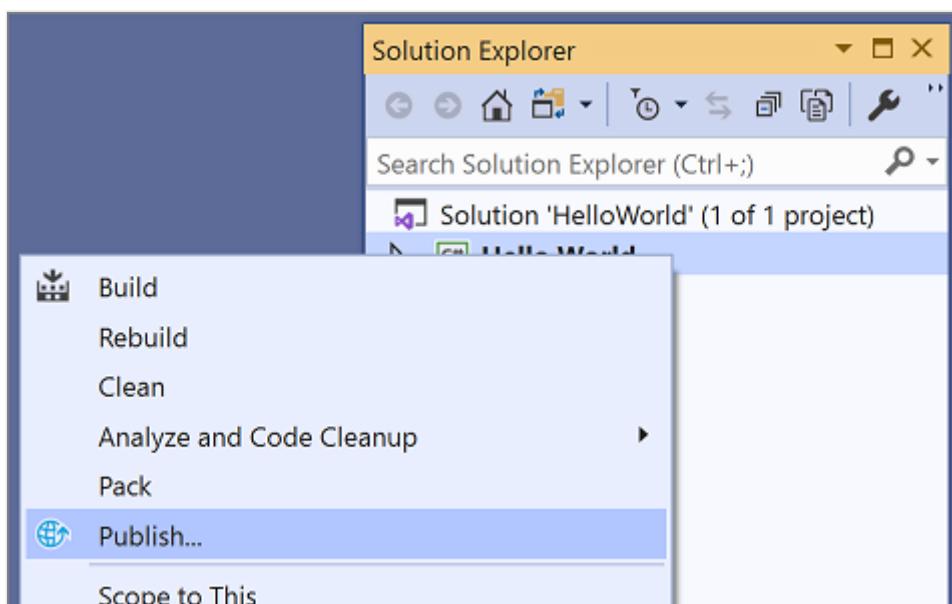
- This tutorial works with the console app that you create in [Create a .NET console application using Visual Studio](#).

Publish the app

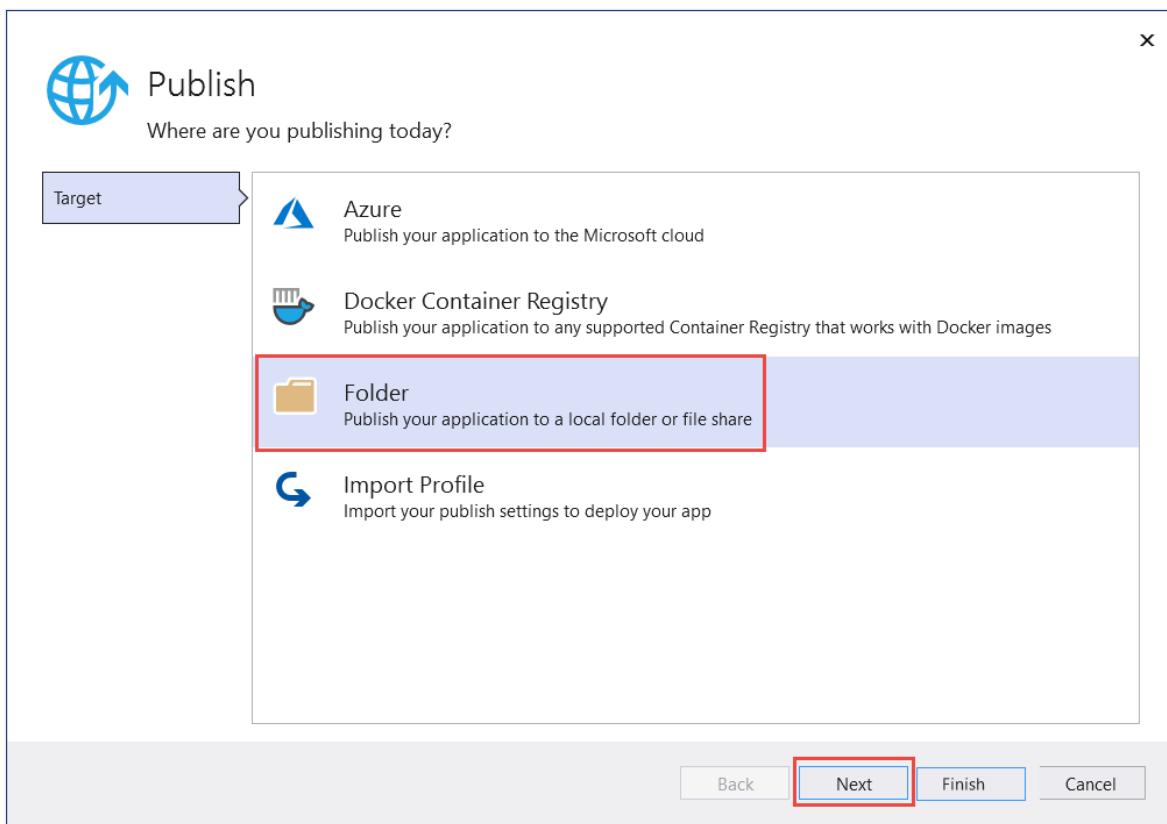
1. Start Visual Studio.
2. Open the *HelloWorld* project that you created in [Create a .NET console application using Visual Studio](#).
3. Make sure that Visual Studio is using the Release build configuration. If necessary, change the build configuration setting on the toolbar from **Debug** to **Release**.



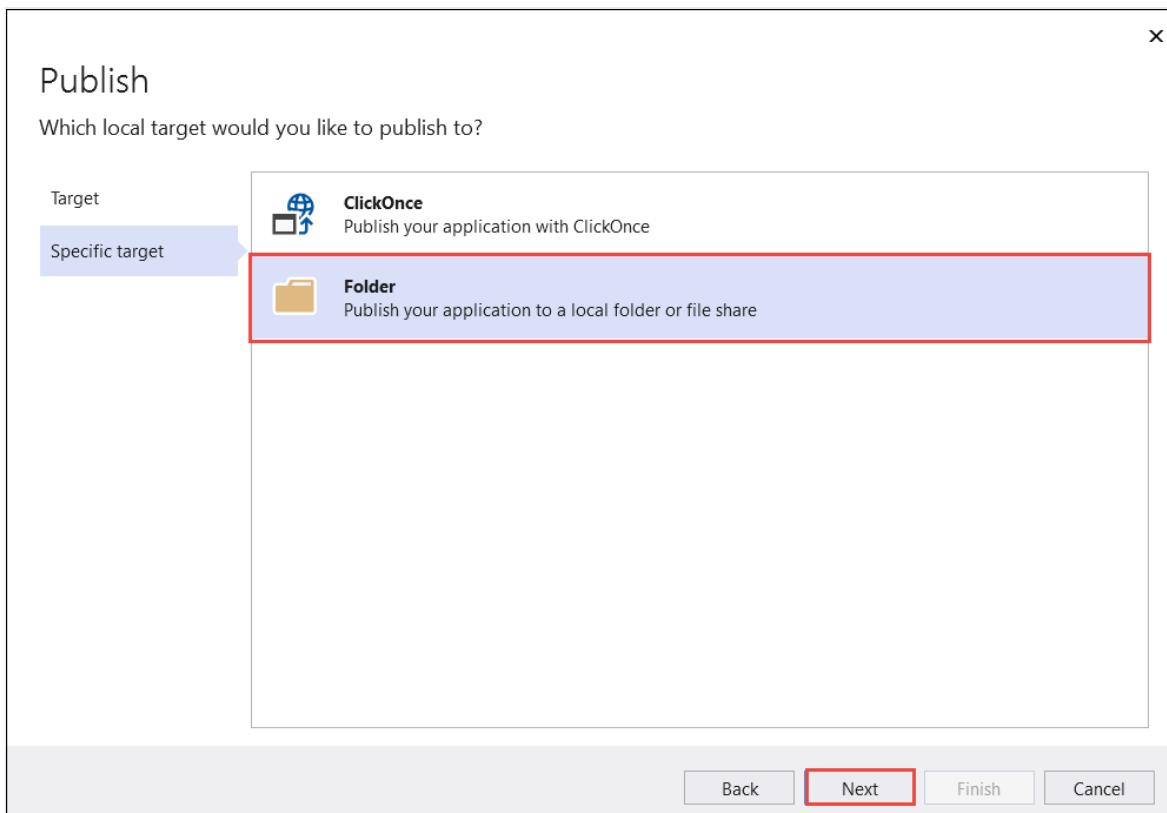
4. Right-click on the **HelloWorld** project (not the **HelloWorld** solution) and select **Publish** from the menu.



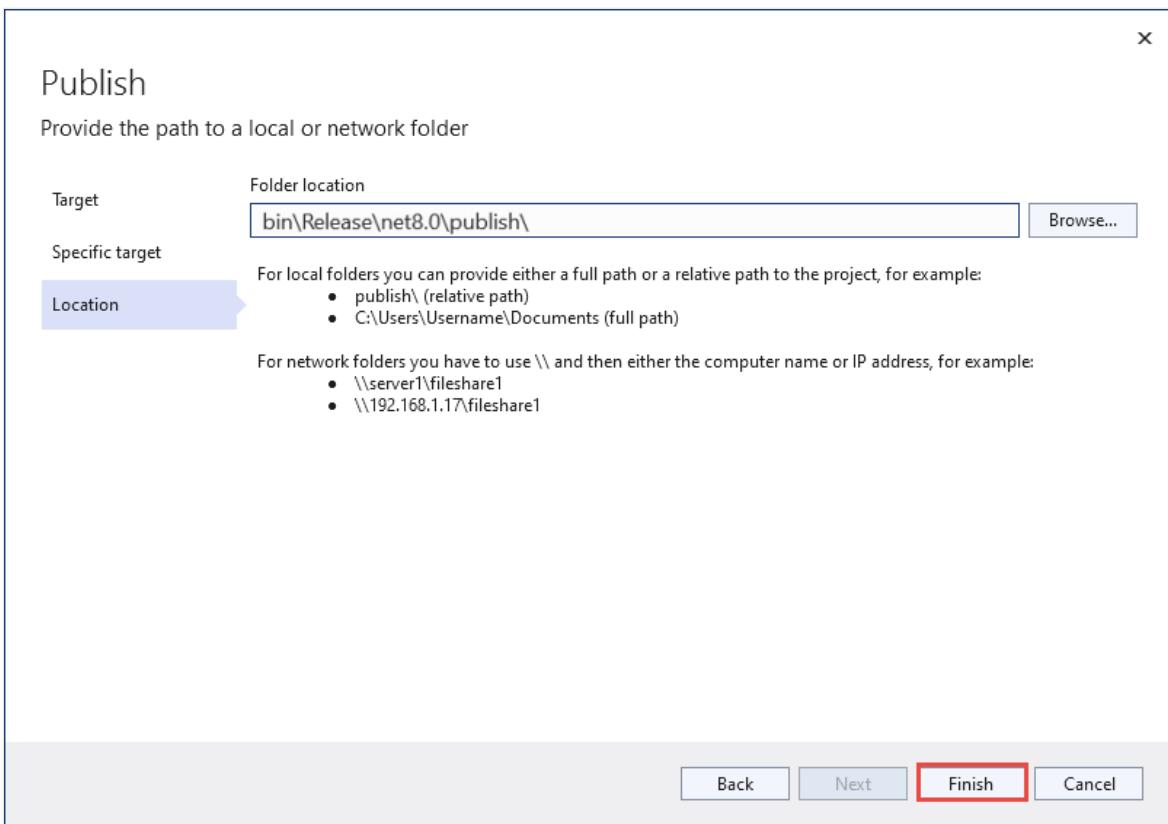
5. On the **Target** tab of the **Publish** page, select **Folder**, and then select **Next**.



6. On the **Specific Target** tab of the **Publish** page, select **Folder**, and then select **Next**.

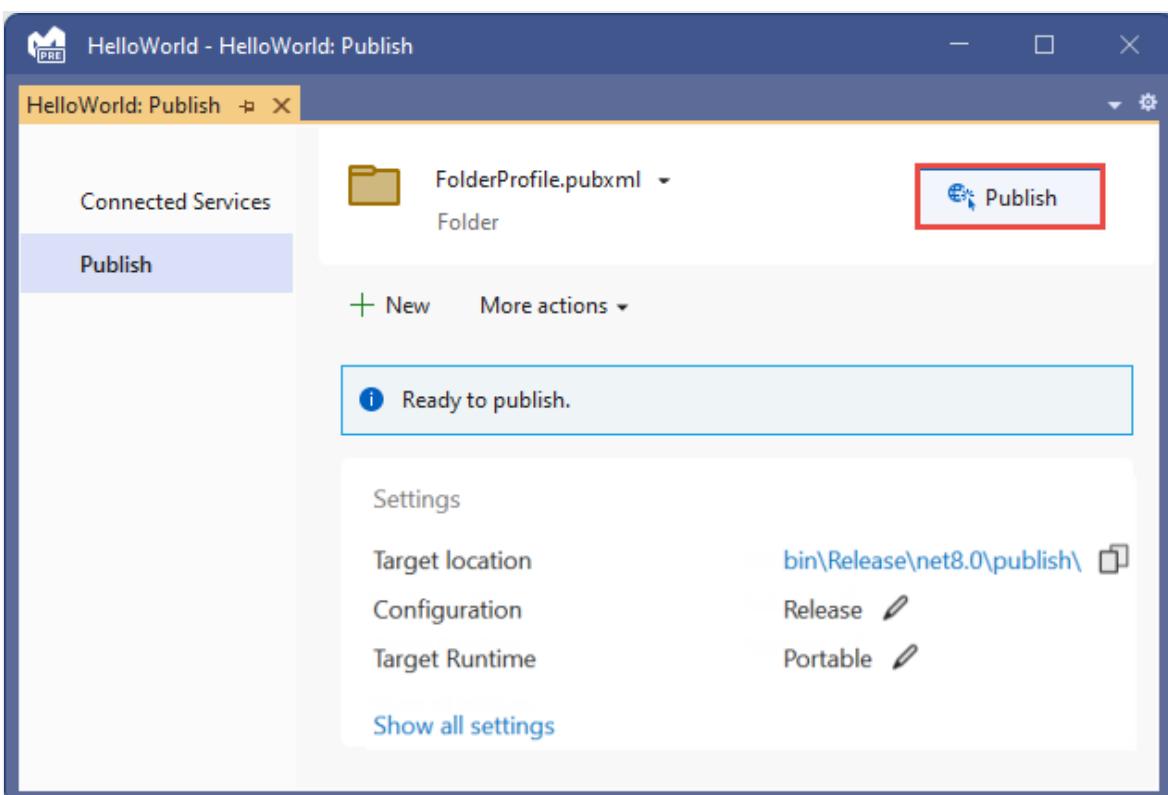


7. On the **Location** tab of the **Publish** page, select **Finish**.



8. On the **Publish profile creation progress** page, select **Close**.

9. On the **Publish** tab of the **Publish** window, select **Publish**.

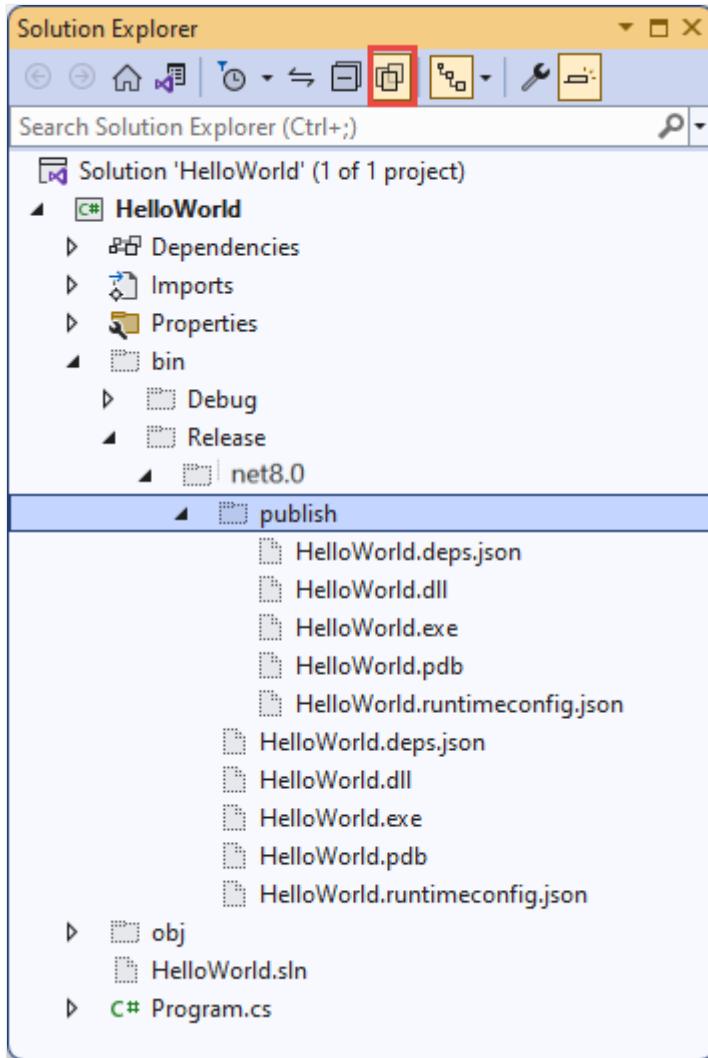


Inspect the files

By default, the publishing process creates a framework-dependent deployment, which is a type of deployment where the published application runs on a machine that has the .NET runtime installed. Users can run the published app by double-clicking the executable or issuing the `dotnet HelloWorld.dll` command from a command prompt.

In the following steps, you'll look at the files created by the publish process.

1. In Solution Explorer, select Show all files.
2. In the project folder, expand `bin/Release/net7.0/publish`.



As the image shows, the published output includes the following files:

- *HelloWorld.deps.json*

This is the application's runtime dependencies file. It defines the .NET components and the libraries (including the dynamic link library that contains your application) needed to run the app. For more information, see [Runtime configuration files ↗](#).

- *HelloWorld.dll*

This is the [framework-dependent deployment](#) version of the application. To execute this dynamic link library, enter `dotnet HelloWorld.dll` at a command prompt. This method of running the app works on any platform that has the .NET runtime installed.

- *HelloWorld.exe*

This is the [framework-dependent executable](#) version of the application. To run it, enter `HelloWorld.exe` at a command prompt. The file is operating-system-specific.

- *HelloWorld.pdb* (optional for deployment)

This is the debug symbols file. You aren't required to deploy this file along with your application, although you should save it in the event that you need to debug the published version of your application.

- *HelloWorld.runtimeconfig.json*

This is the application's runtime configuration file. It identifies the version of .NET that your application was built to run on. You can also add configuration options to it. For more information, see [.NET runtime configuration settings](#).

Run the published app

1. In **Solution Explorer**, right-click the *publish* folder, and select **Copy Full Path**.
2. Open a command prompt and navigate to the *publish* folder. To do that, enter `cd` and then paste the full path. For example:

```
Console
cd C:\Projects\HelloWorld\bin\Release\net8.0\publish\
```

3. Run the app by using the executable:
 - a. Enter `HelloWorld.exe` and press `Enter`.
 - b. Enter a name in response to the prompt, and press any key to exit.
4. Run the app by using the `dotnet` command:
 - a. Enter `dotnet HelloWorld.dll` and press `Enter`.
 - b. Enter a name in response to the prompt, and press any key to exit.

Additional resources

- [.NET application deployment](#)
- [Publish .NET apps with the .NET CLI](#)
- [dotnet publish](#)
- [Tutorial: Publish a .NET console application using Visual Studio Code](#)
- [Use the .NET SDK in continuous integration \(CI\) environments](#)

Next steps

In this tutorial, you published a console app. In the next tutorial, you create a class library.

[Create a .NET class library using Visual Studio](#)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

.NET is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

Tutorial: Create a .NET class library using Visual Studio

Article • 08/25/2023

In this tutorial, you create a simple class library that contains a single string-handling method.

A *class library* defines types and methods that are called by an application. If the library targets .NET Standard 2.0, it can be called by any .NET implementation (including .NET Framework) that supports .NET Standard 2.0. If the library targets .NET 8, it can be called by any application that targets .NET 8. This tutorial shows how to target .NET 8.

When you create a class library, you can distribute it as a NuGet package or as a component bundled with the application that uses it.

Prerequisites

- [Visual Studio 2022 Preview](#) with the **.NET desktop development** workload installed. The .NET 8 SDK is automatically installed when you select this workload.

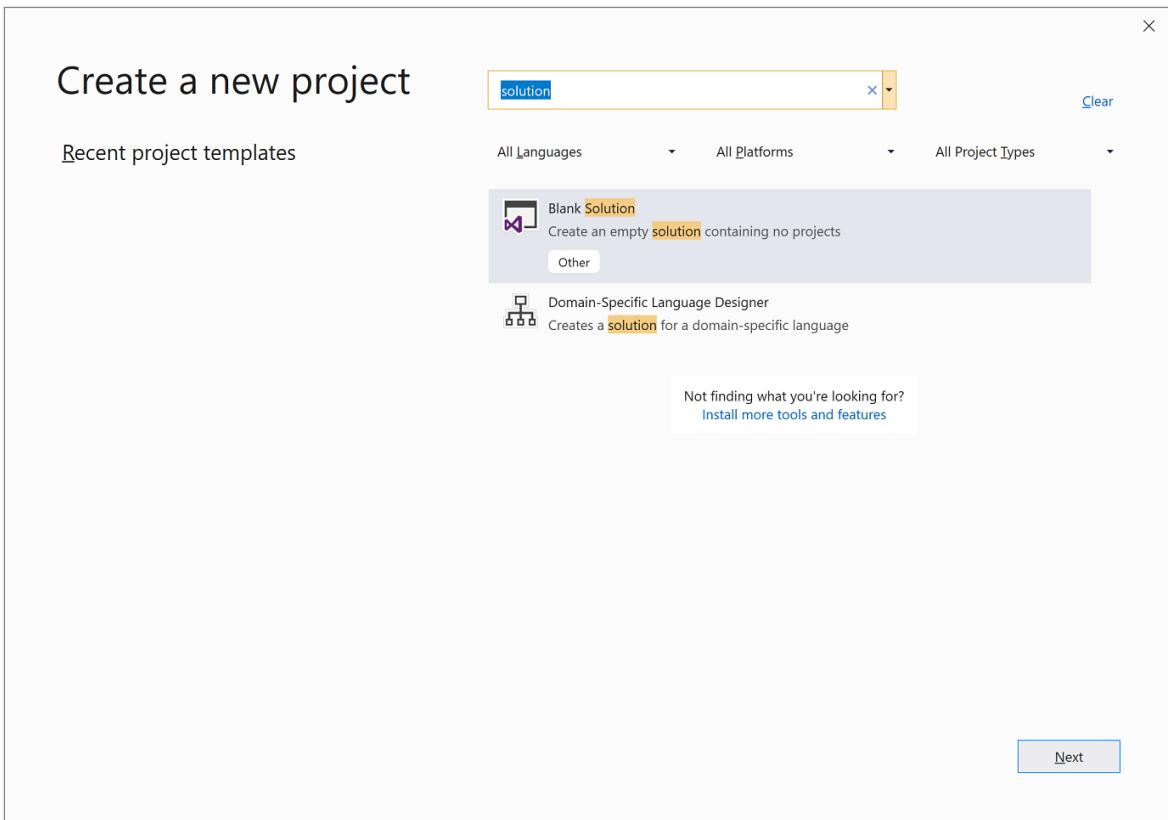
For more information, see [Install the .NET SDK with Visual Studio](#).

Create a solution

Start by creating a blank solution to put the class library project in. A Visual Studio solution serves as a container for one or more projects. You'll add additional, related projects to the same solution.

To create the blank solution:

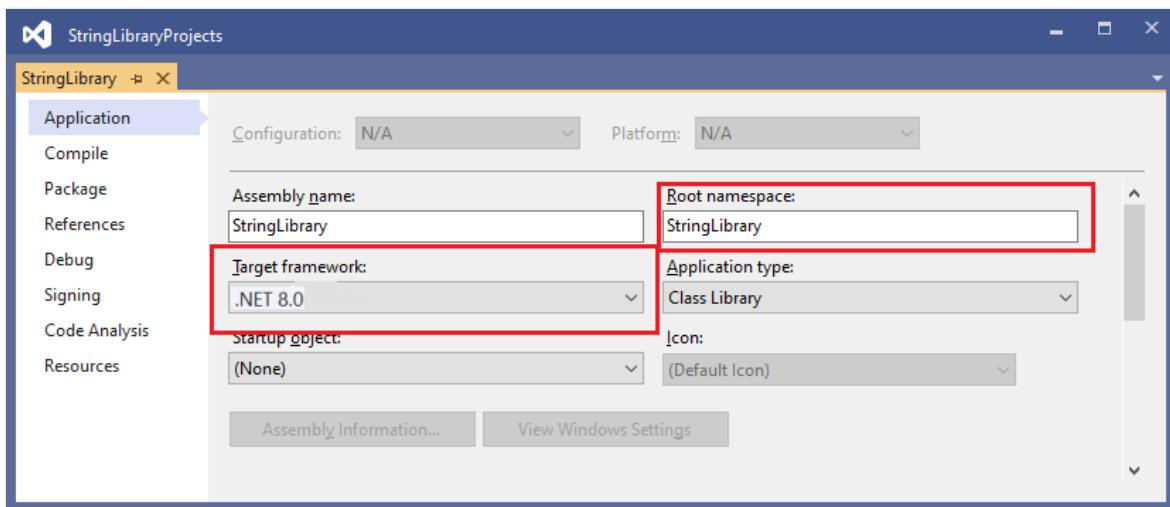
1. Start Visual Studio.
2. On the start window, choose **Create a new project**.
3. On the **Create a new project** page, enter **solution** in the search box. Choose the **Blank Solution** template, and then choose **Next**.



4. On the **Configure your new project** page, enter **ClassLibraryProjects** in the **Solution name** box. Then choose **Create**.

Create a class library project

1. Add a new .NET class library project named "StringLibrary" to the solution.
 - a. Right-click on the solution in **Solution Explorer** and select **Add > New Project**.
 - b. On the **Add a new project** page, enter **library** in the search box. Choose **C#** or **Visual Basic** from the **Language** list, and then choose **All platforms** from the **Platform** list. Choose the **Class Library** template, and then choose **Next**.
 - c. On the **Configure your new project** page, enter **StringLibrary** in the **Project name** box, and then choose **Next**.
 - d. On the **Additional information** page, select **.NET 8 (Preview)**, and then choose **Create**.
2. Check to make sure that the library targets the correct version of .NET. Right-click on the library project in **Solution Explorer**, and then select **Properties**. The **Target Framework** text box shows that the project targets .NET 7.0.
3. If you're using Visual Basic, clear the text in the **Root namespace** text box.



For each project, Visual Basic automatically creates a namespace that corresponds to the project name. In this tutorial, you define a top-level namespace by using the [namespace](#) keyword in the code file.

4. Replace the code in the code window for *Class1.cs* or *Class1.vb* with the following code, and save the file. If the language you want to use isn't shown, change the language selector at the top of the page.

```
C#  
  
namespace UtilityLibraries;  
  
public static class StringLibrary  
{  
    public static bool StartsWithUpper(this string? str)  
    {  
        if (string.IsNullOrWhiteSpace(str))  
            return false;  
  
        char ch = str[0];  
        return char.IsUpper(ch);  
    }  
}
```

The class library, `UtilityLibraries.StringLibrary`, contains a method named `StartsWithUpper`. This method returns a [Boolean](#) value that indicates whether the current string instance begins with an uppercase character. The Unicode standard distinguishes uppercase characters from lowercase characters. The `Char.IsUpper(Char)` method returns `true` if a character is uppercase.

`StartsWithUpper` is implemented as an [extension method](#) so that you can call it as if it were a member of the `String` class. The question mark (?) after `string` in the C# code indicates that the string may be null.

5. On the menu bar, select **Build > Build Solution** or press **Ctrl + Shift + B** to verify that the project compiles without error.

Add a console app to the solution

Add a console application that uses the class library. The app will prompt the user to enter a string and report whether the string begins with an uppercase character.

1. Add a new .NET console application named "ShowCase" to the solution.
 - a. Right-click on the solution in **Solution Explorer** and select **Add > New project**.
 - b. On the **Add a new project** page, enter **console** in the search box. Choose **C#** or **Visual Basic** from the Language list, and then choose **All platforms** from the Platform list.
 - c. Choose the **Console Application** template, and then choose **Next**.
 - d. On the **Configure your new project** page, enter **ShowCase** in the **Project name** box. Then choose **Next**.
 - e. On the **Additional information** page, select **.NET 8 (Preview)** in the **Framework** box. Then choose **Create**.
2. In the code window for the *Program.cs* or *Program.vb* file, replace all of the code with the following code.

```
C#  
  
using UtilityLibraries;  
  
class Program  
{  
    static void Main(string[] args)  
    {  
        int row = 0;  
  
        do  
        {  
            if (row == 0 || row >= 25)  
                ResetConsole();  
  
            string? input = Console.ReadLine();  
            if (string.IsNullOrEmpty(input)) break;  
            Console.WriteLine($"Input: {input}");  
            Console.WriteLine("Begins with uppercase? " +  
                $"{(input.StartsWithUpper() ? "Yes" : "No")});  
            Console.WriteLine();
```

```
        row += 4;
    } while (true);
    return;

    // Declare a ResetConsole local method
    void ResetConsole()
    {
        if (row > 0)
        {
            Console.WriteLine("Press any key to continue...");
            Console.ReadKey();
        }
        Console.Clear();
        Console.WriteLine($"{Environment.NewLine}Press <Enter> only
to exit; otherwise, enter a string and press <Enter>:
{Environment.NewLine}");
        row = 3;
    }
}
```

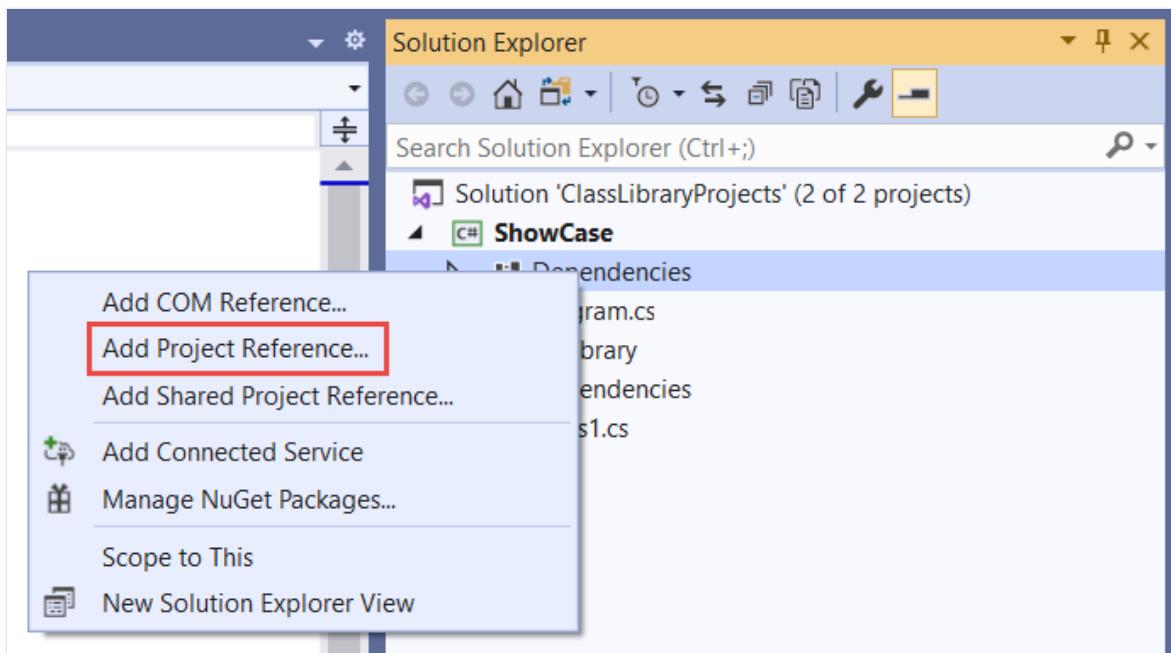
The code uses the `row` variable to maintain a count of the number of rows of data written to the console window. Whenever it's greater than or equal to 25, the code clears the console window and displays a message to the user.

The program prompts the user to enter a string. It indicates whether the string starts with an uppercase character. If the user presses the `Enter` key without entering a string, the application ends, and the console window closes.

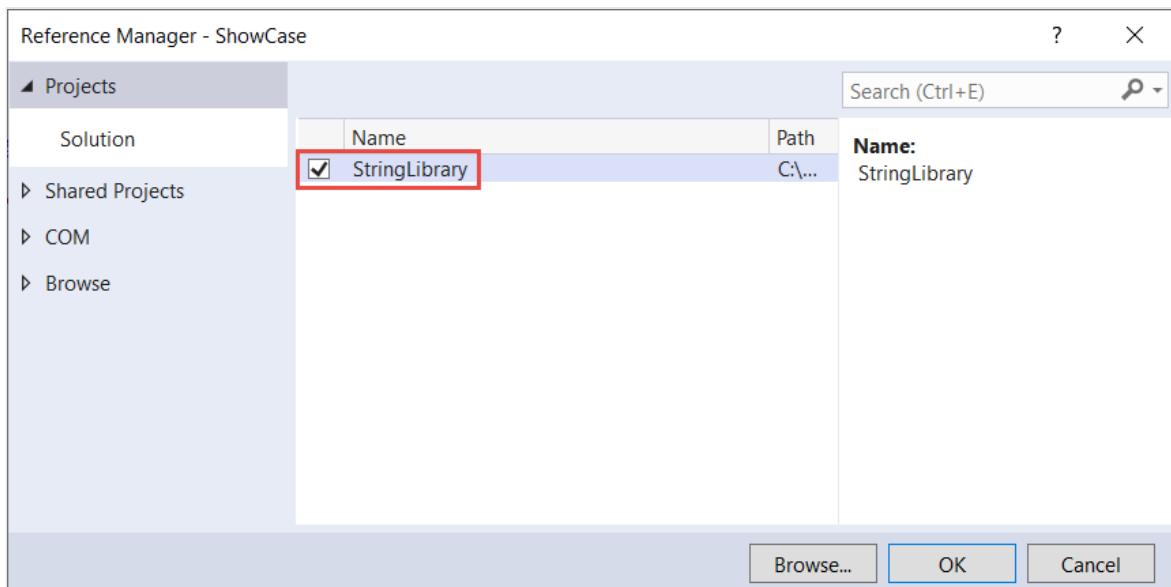
Add a project reference

Initially, the new console app project doesn't have access to the class library. To allow it to call methods in the class library, create a project reference to the class library project.

1. In **Solution Explorer**, right-click the `ShowCase` project's **Dependencies** node, and select **Add Project Reference**.

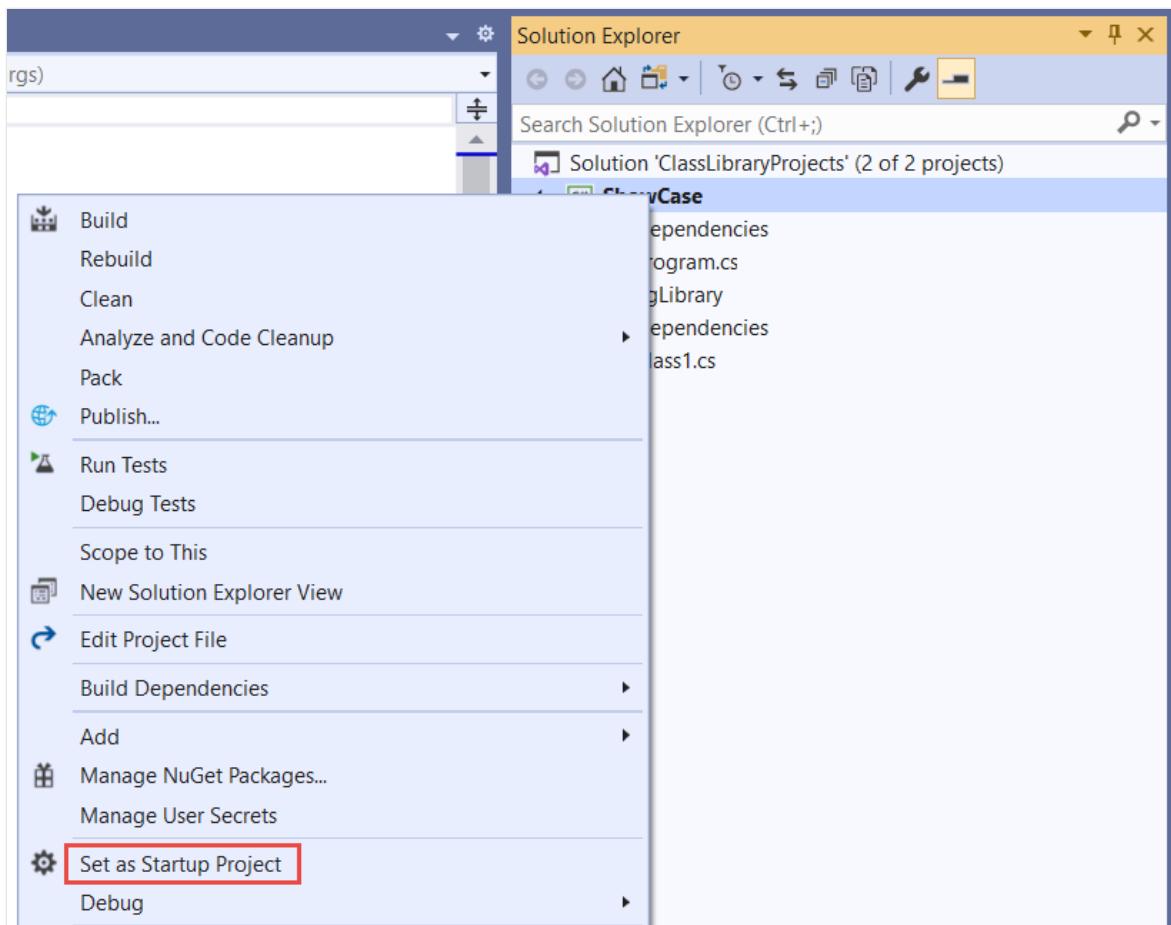


2. In the Reference Manager dialog, select the StringLibrary project, and select OK.



Run the app

1. In Solution Explorer, right-click the Showcase project and select Set as StartUp Project in the context menu.



2. Press **Ctrl + F5** to compile and run the program without debugging.
3. Try out the program by entering strings and pressing **Enter**, then press **Enter** to exit.

```
Press <Enter> only to exit; otherwise, enter a string and press <Enter>:  
Hello  
Input: Hello  
Begins with uppercase? Yes  
  
hello  
Input: hello  
Begins with uppercase? No
```

Additional resources

- Develop libraries with the .NET CLI
- .NET Standard versions and the platforms they support.

Next steps

In this tutorial, you created a class library. In the next tutorial, you learn how to unit test the class library.

Unit test a .NET class library using Visual Studio

Or you can skip automated unit testing and learn how to share the library by creating a NuGet package:

Create and publish a package using Visual Studio

Or learn how to publish a console app. If you publish the console app from the solution you created in this tutorial, the class library goes with it as a *.dll* file.

Publish a .NET console application using Visual Studio

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

Tutorial: Test a .NET class library with .NET using Visual Studio

Article • 08/25/2023

This tutorial shows how to automate unit testing by adding a test project to a solution.

Prerequisites

- This tutorial works with the solution that you create in [Create a .NET class library using Visual Studio](#).

Create a unit test project

Unit tests provide automated software testing during your development and publishing. [MSTest](#) is one of three test frameworks you can choose from. The others are [xUnit](#) and [nUnit](#).

1. Start Visual Studio.
2. Open the `ClassLibraryProjects` solution you created in [Create a .NET class library using Visual Studio](#).
3. Add a new unit test project named "StringLibraryTest" to the solution.
 - a. Right-click on the solution in **Solution Explorer** and select **Add > New project**.
 - b. On the **Add a new project** page, enter **mstest** in the search box. Choose **C#** or **Visual Basic** from the Language list, and then choose **All platforms** from the Platform list.
 - c. Choose the **MSTest Test Project** template, and then choose **Next**.
 - d. On the **Configure your new project** page, enter **StringLibraryTest** in the **Project name** box. Then choose **Next**.
 - e. On the **Additional information** page, select **.NET 8 (Preview)** in the **Framework** box. Then choose **Create**.
4. Visual Studio creates the project and opens the class file in the code window with the following code. If the language you want to use is not shown, change the language selector at the top of the page.

C#

```
namespace StringLibraryTest;

[TestClass]
public class UnitTest1
{
    [TestMethod]
    public void TestMethod1()
    {
    }
}
```

The source code created by the unit test template does the following:

- It imports the `Microsoft.VisualStudio.TestTools.UnitTesting` namespace, which contains the types used for unit testing. In C#, the namespace is imported via a `global using` directive in `GlobalUsings.cs`.
- It applies the `TestClassAttribute` attribute to the `UnitTest1` class.
- It applies the `TestMethodAttribute` attribute to define `TestMethod1` in C# or `TestSub` in Visual Basic.

Each method tagged with `[TestMethod]` in a test class tagged with `[TestClass]` is executed automatically when the unit test is run.

Add a project reference

For the test project to work with the `StringLibrary` class, add a reference in the `StringLibraryTest` project to the `StringLibrary` project.

1. In **Solution Explorer**, right-click the **Dependencies** node of the `StringLibraryTest` project and select **Add Project Reference** from the context menu.
2. In the **Reference Manager** dialog, expand the **Projects** node, and select the box next to `StringLibrary`. Adding a reference to the `StringLibrary` assembly allows the compiler to find `StringLibrary` methods while compiling the `StringLibraryTest` project.
3. Select **OK**.

Add and run unit test methods

When Visual Studio runs a unit test, it executes each method that is marked with the [TestMethodAttribute](#) attribute in a class that is marked with the [TestClassAttribute](#) attribute. A test method ends when the first failure is found or when all tests contained in the method have succeeded.

The most common tests call members of the [Assert](#) class. Many assert methods include at least two parameters, one of which is the expected test result and the other of which is the actual test result. Some of the [Assert](#) class's most frequently called methods are shown in the following table:

| Assert methods | Function |
|-------------------------------|--|
| <code>Assert.AreEqual</code> | Verifies that two values or objects are equal. The assert fails if the values or objects aren't equal. |
| <code>Assert.AreSame</code> | Verifies that two object variables refer to the same object. The assert fails if the variables refer to different objects. |
| <code>Assert.IsFalse</code> | Verifies that a condition is <code>false</code> . The assert fails if the condition is <code>true</code> . |
| <code>Assert.IsNotNull</code> | Verifies that an object isn't <code>null</code> . The assert fails if the object is <code>null</code> . |

You can also use the [Assert.ThrowsException](#) method in a test method to indicate the type of exception it's expected to throw. The test fails if the specified exception isn't thrown.

In testing the `StringLibrary.StartsWithUpper` method, you want to provide a number of strings that begin with an uppercase character. You expect the method to return `true` in these cases, so you can call the [Assert.IsTrue](#) method. Similarly, you want to provide a number of strings that begin with something other than an uppercase character. You expect the method to return `false` in these cases, so you can call the [Assert.IsFalse](#) method.

Since your library method handles strings, you also want to make sure that it successfully handles an [empty string \(String.Empty\)](#), a valid string that has no characters and whose [Length](#) is 0, and a `null` string that hasn't been initialized. You can call `StartsWithUpper` directly as a static method and pass a single [String](#) argument. Or you can call `StartsWithUpper` as an extension method on a `string` variable assigned to `null`.

You'll define three methods, each of which calls an [Assert](#) method for each element in a string array. You'll call a method overload that lets you specify an error message to be displayed in case of test failure. The message identifies the string that caused the failure.

To create the test methods:

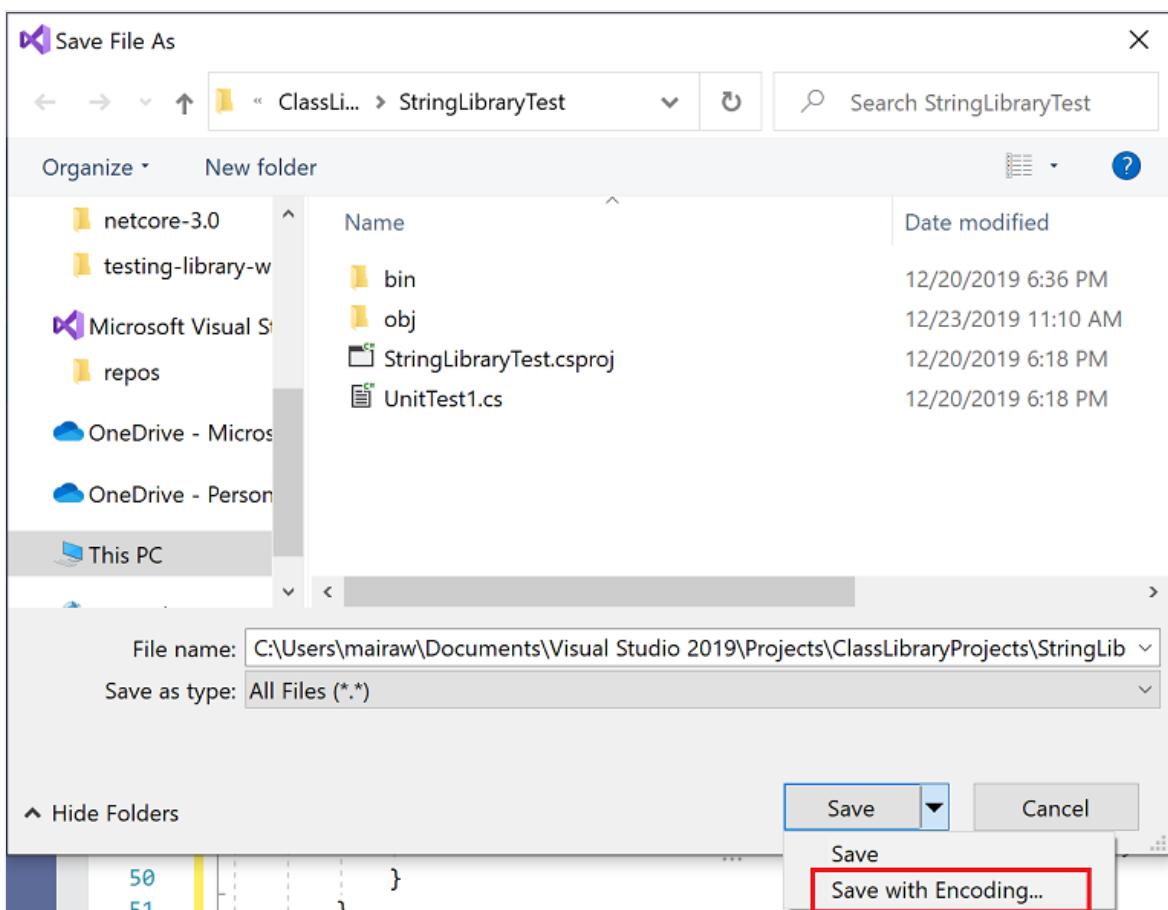
1. In the *UnitTest1.cs* or *UnitTest1.vb* code window, replace the code with the following code:

```
C#  
  
using UtilityLibraries;  
  
namespace StringLibraryTest  
{  
    [TestClass]  
    public class UnitTest1  
    {  
        [TestMethod]  
        public void TestStartsWithUpper()  
        {  
            // Tests that we expect to return true.  
            string[] words = { "Alphabet", "Zebra", "ABC", "Αθήνα",  
"Москва" };  
            foreach (var word in words)  
            {  
                bool result = word.StartsWithUpper();  
                Assert.IsTrue(result,  
                    string.Format("Expected for '{0}': true; Actual:  
{1}",  
                                word, result));  
            }  
        }  
  
        [TestMethod]  
        public void TestDoesNotStartWithUpper()  
        {  
            // Tests that we expect to return false.  
            string[] words = { "alphabet", "zebra", "abc",  
"αυτοκινητοβιομηχανία", "государство",  
                    "1234", ".", ";", " " };  
            foreach (var word in words)  
            {  
                bool result = word.StartsWithUpper();  
                Assert.IsFalse(result,  
                    string.Format("Expected for '{0}': false;  
Actual: {1}",  
                                word, result));  
            }  
        }  
  
        [TestMethod]  
        public void DirectCallWithNullOrEmpty()  
        {  
            // Tests that we expect to return false.  
            string?[] words = { string.Empty, null };  
            foreach (var word in words)
```

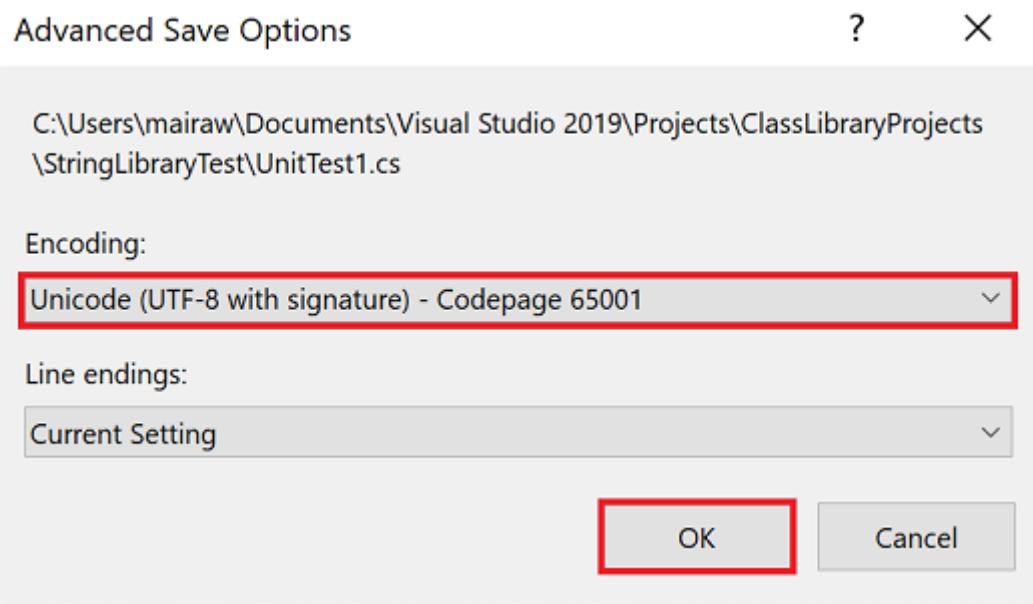
```
        {
            bool result = StringLibrary.StartsWithUpper(word);
            Assert.IsFalse(result,
                string.Format("Expected for '{0}': false;
Actual: {1}", word == null ? "<null>" : word,
result));
        }
    }
}
```

The test of uppercase characters in the `TestStartsWithUpper` method includes the Greek capital letter alpha (U+0391) and the Cyrillic capital letter EM (U+041C). The test of lowercase characters in the `TestDoesNotStartWithUpper` method includes the Greek small letter alpha (U+03B1) and the Cyrillic small letter Ghe (U+0433).

2. On the menu bar, select **File > Save UnitTest1.cs As** or **File > Save UnitTest1.vb As**. In the **Save File As** dialog, select the arrow beside the **Save** button, and select **Save with Encoding**.

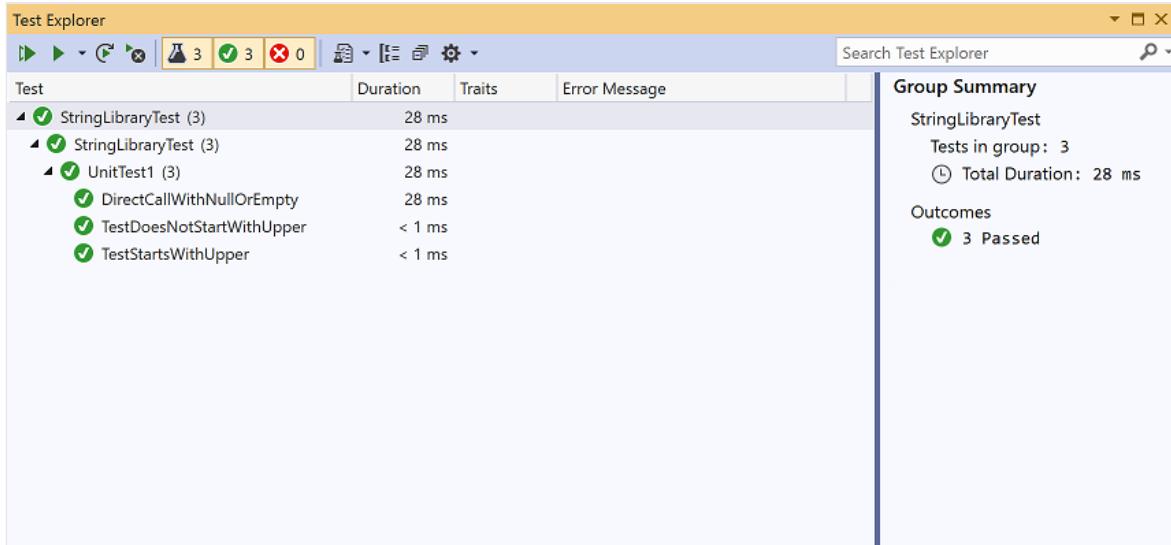


3. In the **Confirm Save As** dialog, select the **Yes** button to save the file.
 4. In the **Advanced Save Options** dialog, select **Unicode (UTF-8 with signature) - Codepage 65001** from the **Encoding** drop-down list and select **OK**.



If you fail to save your source code as a UTF8-encoded file, Visual Studio may save it as an ASCII file. When that happens, the runtime doesn't accurately decode the UTF8 characters outside of the ASCII range, and the test results won't be correct.

5. On the menu bar, select **Test** > **Run All Tests**. If the **Test Explorer** window doesn't open, open it by choosing **Test** > **Test Explorer**. The three tests are listed in the **Passed Tests** section, and the **Summary** section reports the result of the test run.



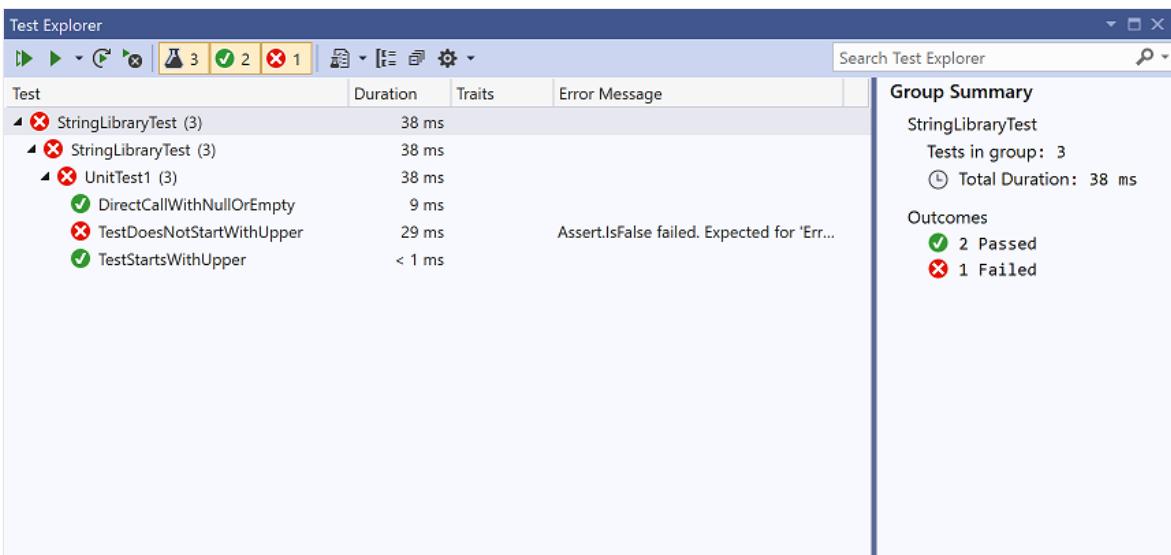
Handle test failures

If you're doing test-driven development (TDD), you write tests first and they fail the first time you run them. Then you add code to the app that makes the test succeed. For this tutorial, you created the test after writing the app code that it validates, so you haven't seen the test fail. To validate that a test fails when you expect it to fail, add an invalid value to the test input.

1. Modify the `words` array in the `TestDoesNotStartWithUpper` method to include the string "Error". You don't need to save the file because Visual Studio automatically saves open files when a solution is built to run tests.

```
C#  
  
string[] words = { "alphabet", "Error", "zebra", "abc",  
    "αυτοκινητοβιομηχανία", "государство",  
    "1234", ".", ";", " " };
```

2. Run the test by selecting **Test > Run All Tests** from the menu bar. The **Test Explorer** window indicates that two tests succeeded and one failed.



3. Select the failed test, `TestDoesNotStartWithUpper`.

The **Test Explorer** window displays the message produced by the assert: "Assert.IsFalse failed. Expected for 'Error': false; actual: True". Because of the failure, no strings in the array after "Error" were tested.

Test Detail Summary

✗ TestDoesNotStartWithUpper

Source: [UnitTest1.cs](#) line 25

Duration: 29 ms

Message:

```
Assert.IsFalse failed. Expected for 'Error': false; Actual: True
```

Stack Trace:

```
UnitTest1.TestDoesNotStartWithUpper() line 34
```

4. Remove the string "Error" that you added in step 1. Rerun the test and the tests pass.

Test the Release version of the library

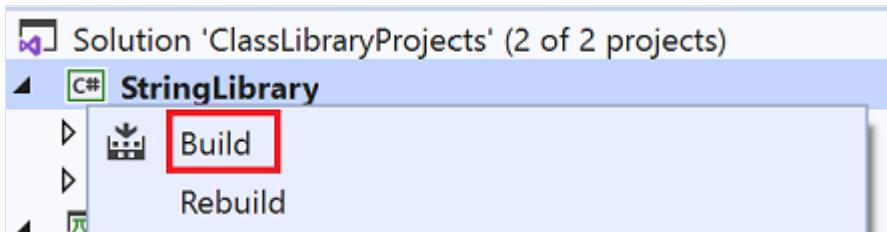
Now that the tests have all passed when running the Debug build of the library, run the tests an additional time against the Release build of the library. A number of factors, including compiler optimizations, can sometimes produce different behavior between Debug and Release builds.

To test the Release build:

1. In the Visual Studio toolbar, change the build configuration from **Debug** to **Release**.



2. In **Solution Explorer**, right-click the **StringLibrary** project and select **Build** from the context menu to recompile the library.



3. Run the unit tests by choosing **Test > Run All Tests** from the menu bar. The tests pass.

Debug tests

If you're using Visual Studio as your IDE, you can use the same process shown in [Tutorial: Debug a .NET console application using Visual Studio](#) to debug code using your unit test project. Instead of starting the *ShowCase* app project, right-click the **StringLibraryTests** project, and select **Debug Tests** from the context menu.

Visual Studio starts the test project with the debugger attached. Execution will stop at any breakpoint you've added to the test project or the underlying library code.

Additional resources

- [Unit test basics - Visual Studio](#)
- [Unit testing in .NET](#)

Next steps

In this tutorial, you unit tested a class library. You can make the library available to others by publishing it to [NuGet](#) as a package. To learn how, follow a NuGet tutorial:

[Create and publish a NuGet package using Visual Studio](#)

If you publish a library as a NuGet package, others can install and use it. To learn how, follow a NuGet tutorial:

[Install and use a package in Visual Studio](#)

A library doesn't have to be distributed as a package. It can be bundled with a console app that uses it. To learn how to publish a console app, see the earlier tutorial in this series:

[Publish a .NET console application using Visual Studio](#)

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback [here](#).

 [Open a documentation issue](#)

 [Provide product feedback](#)

Quickstart: Install and use a NuGet package in Visual Studio (Windows only)

Article • 08/21/2023

A *NuGet package* contains reusable code that other developers have made available to you for use in your projects. You can install a NuGet package in a Microsoft Visual Studio project by using the [NuGet Package Manager](#), the [Package Manager Console](#), or the [.NET CLI](#). This article demonstrates how to create a Windows Presentation Foundation (WPF) project with the popular `Newtonsoft.Json` package. The same process applies to any other .NET or .NET Core project.

After you install a NuGet package, you can then make a reference to it in your code with the `using <namespace>` statement, where `<namespace>` is the name of package you're using. After you've made a reference, you can then call the package through its API.

The article is for Windows users only. If you're using Visual Studio for Mac, see [Install and use a package in Visual Studio for Mac](#).

Tip

To find a NuGet package, start with [nuget.org](#). Browsing nuget.org is how .NET developers typically find components they can reuse in their own applications. You can do a search of nuget.org directly or find and install packages within Visual Studio as shown in this article. For more information, see [Find and evaluate NuGet packages](#).

Prerequisites

- Install Visual Studio 2022 for Windows with the .NET desktop development workload.

You can install the 2022 Community edition for free from visualstudio.microsoft.com , or use the Professional or Enterprise edition.

Create a project

You can install a NuGet package into any .NET project if that package supports the same target framework as the project. However, for this quickstart you'll create a Windows Presentation Foundation (WPF) Application project.

Follow these steps:

1. In Visual Studio, select **File > New > Project**.
2. In the **Create a new project** window, enter **WPF** in the search box and select **C#** and **Windows** in the dropdown lists. In the resulting list of project templates, select **WPF Application**, and then select **Next**.
3. In the **Configure your new project** window, optionally update the **Project name** and the **Solution name**, and then select **Next**.
4. In the **Additional information** window, select **.NET 6.0** (or the latest version) for **Framework**, and then select **Create**.

Visual Studio creates the project, and it appears in [Solution Explorer](#).

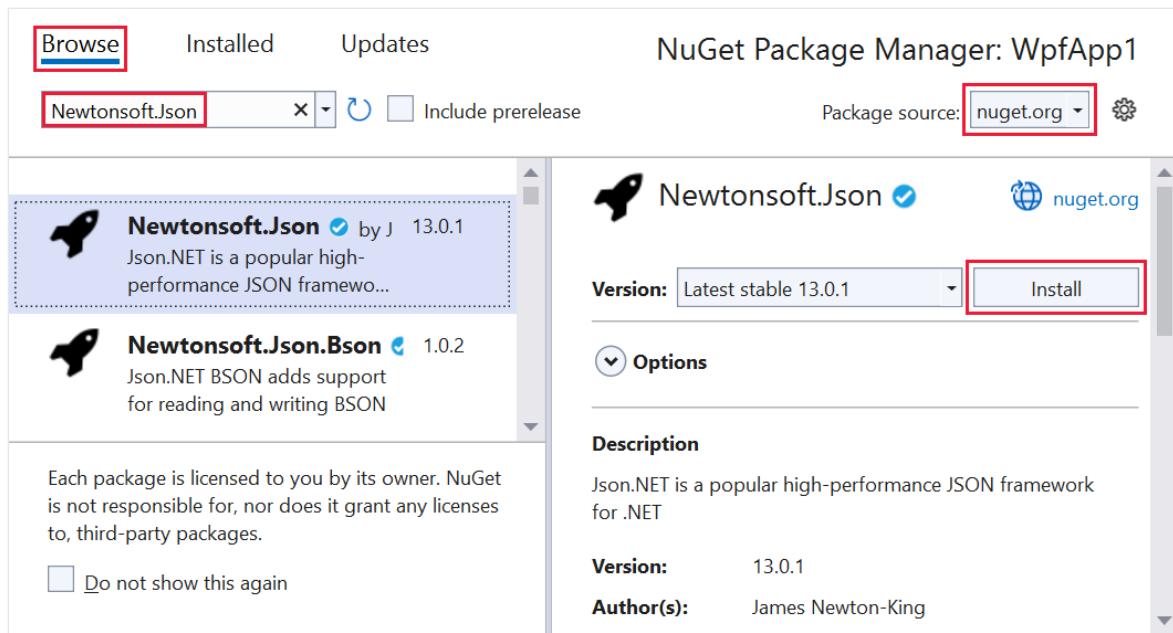
Add the Newtonsoft.Json NuGet package

To install a NuGet package in this quickstart, you can use either the NuGet Package Manager or the Package Manager Console. Depending on your project format, the installation of a NuGet package records the dependency in either your project file or a `packages.config` file. For more information, see [Package consumption workflow](#).

NuGet Package Manager

To use the [NuGet Package Manager](#) to install the `Newtonsoft.Json` package in Visual Studio, follow these steps:

1. Select **Project > Manage NuGet Packages**.
2. In the **NuGet Package Manager** page, choose **nuget.org** as the **Package source**.
3. From the **Browse** tab, search for `Newtonsoft.Json`, select **Newtonsoft.Json** in the list, and then select **Install**.

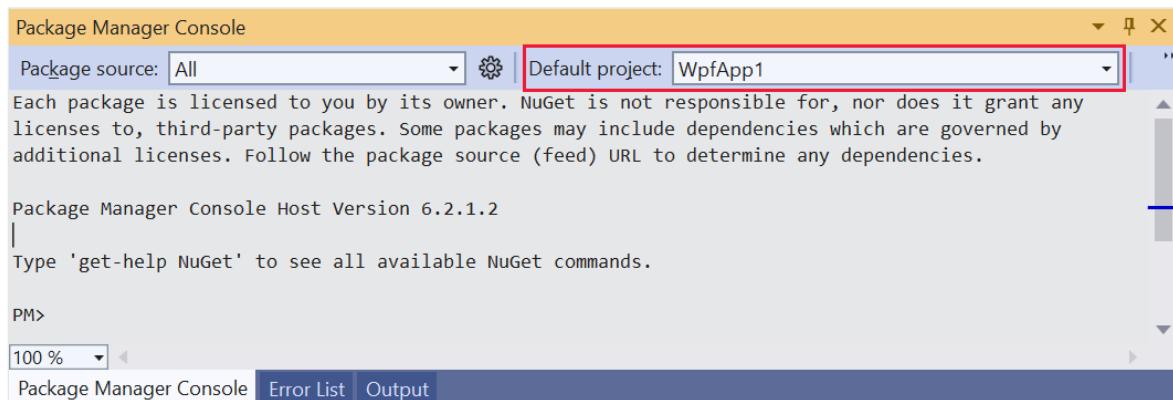


4. If you're prompted to verify the installation, select **OK**.

Package Manager Console

Alternatively, to use the [Package Manager Console](#) in Visual Studio to install the `Newtonsoft.Json` package, follow these steps:

1. From Visual Studio, select **Tools > NuGet Package Manager > Package Manager Console**.
2. After the **Package Manager Console** pane opens, verify that the **Default project** drop-down list shows the project in which you want to install the package. If you have a single project in the solution, it's preselected.



3. At the console prompt, enter the command `Install-Package Newtonsoft.Json`. For more information about this command, see [Install-Package](#).

The console window shows the output for the command. Errors typically indicate that the package isn't compatible with the project's target framework.

Use the Newtonsoft.Json API in the app

With the `Newtonsoft.Json` package in the project, call its `JsonConvert.SerializeObject` method to convert an object to a human-readable string:

1. From **Solution Explorer**, open *MainWindow.xaml* and replace the existing `<Grid>` element with the following code:

XAML

```
<Grid Background="White">
    <StackPanel VerticalAlignment="Center">
        <Button Click="Button_Click" Width="100px"
HorizontalAlignment="Center" Content="Click Me" Margin="10"/>
        <TextBlock Name="TextBlock" HorizontalAlignment="Center"
Text="TextBlock" Margin="10"/>
    </StackPanel>
</Grid>
```

2. Open the *MainWindow.xaml.cs* file under the *MainWindow.xaml* node, and insert the following code inside the `MainWindow` class after the constructor:

C#

```
public class Account
{
    public string Name { get; set; }
    public string Email { get; set; }
    public DateTime DOB { get; set; }
}

private void Button_Click(object sender, RoutedEventArgs e)
{
    Account account = new Account
    {
        Name = "John Doe",
        Email = "john@microsoft.com",
        DOB = new DateTime(1980, 2, 20, 0, 0, 0, DateTimeKind.Utc),
    };
    string json = JsonConvert.SerializeObject(account,
Newtonsoft.Json.Formatting.Indented);
    TextBlock.Text = json;
}
```

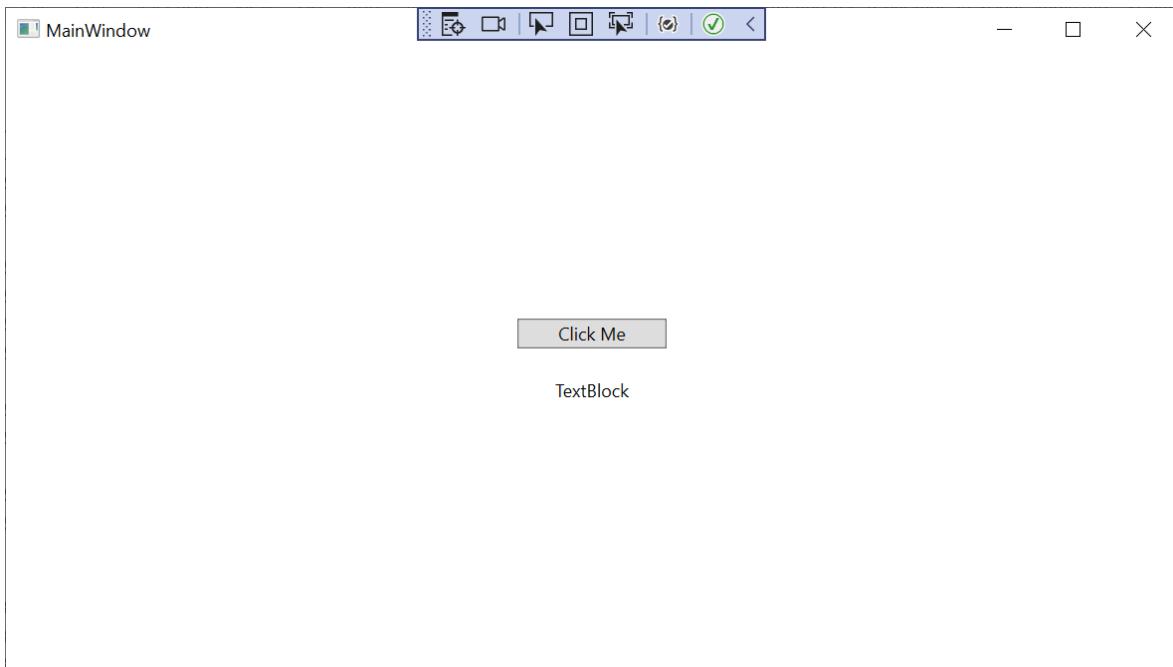
3. To avoid an error for the `JsonConvert` object in the code (a red squiggle line will appear), add the following statement at the beginning of the code file:

C#

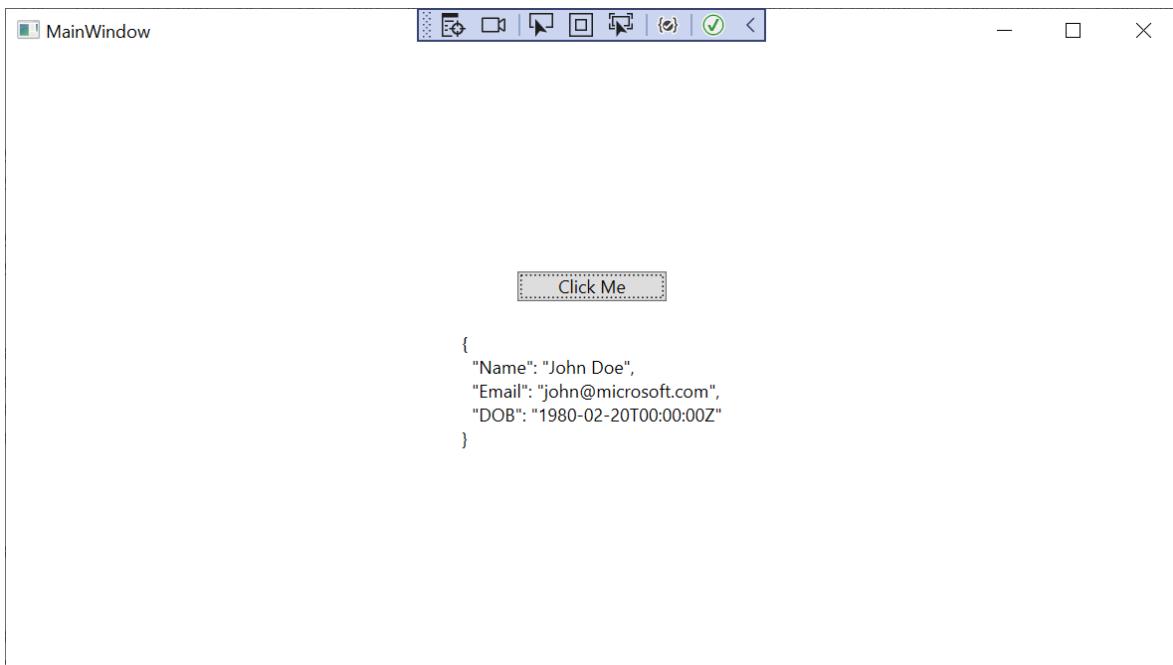
```
using Newtonsoft.Json;
```

4. To build and run the app, press F5 or select **Debug > Start Debugging**.

The following window appears:



5. Select the **Click Me** button to see the contents of the `TextBlock` object replaced with JSON text.



Related video

- [Install and Use a NuGet Package with Visual Studio](#)

- Find more NuGet videos on [Channel 9](#) and [YouTube](#).

See also

For more information about NuGet, see the following articles:

- [What is NuGet?](#)
- [Package consumption workflow](#)
- [Find and choose packages](#)
- [Package references in project files](#)
- [Install and use a package using the .NET CLI.](#)
- [Newtonsoft.Json package](#)

Next steps

Congratulations on installing and using your first NuGet package. Advance to the next article to learn more about installing and managing NuGet packages.

[Install and manage packages using using the NuGet Package Manager](#)

[Install and manage packages using the Package Manager Console](#)

Quickstart: Create and publish a NuGet package using Visual Studio (Windows only)

Article • 08/21/2023

With Microsoft Visual Studio, you can create a NuGet package from a .NET class library, and then publish it to nuget.org using a CLI tool.

The quickstart is for Windows users only. If you're using Visual Studio for Mac, see [Create a NuGet package from existing library projects](#) or use the [.NET CLI](#).

Prerequisites

- Install Visual Studio 2022 for Windows with a .NET Core-related workload.

You can install the 2022 Community edition for free from visualstudio.microsoft.com, or use the Professional or Enterprise edition.

Visual Studio 2017 and later automatically includes NuGet capabilities when you install a .NET-related workload.

- Install the .NET CLI, if it's not already installed.

For Visual Studio 2017 and later, the .NET CLI is automatically installed with any .NET Core-related workload. Otherwise, install the [.NET Core SDK](#) to get the .NET CLI. The .NET CLI is required for .NET projects that use the [SDK-style format](#) (SDK attribute). The default .NET class library template in Visual Studio 2017 and later uses the SDK attribute.

Important

If you're working with a non-SDK-style project, follow the procedures in [Create and publish a .NET Framework package \(Visual Studio\)](#) instead to create and publish the package. For this article, the .NET CLI is recommended. Although you can publish any NuGet package using the NuGet CLI, some of the steps in this article are specific to SDK-style projects and the .NET CLI. The NuGet CLI is used for [non-SDK-style projects](#) (typically .NET Framework).

- Register for a free account on [nuget.org](#) if you don't have one already. You must register and confirm the account before you can upload a NuGet package.
- Install the NuGet CLI by downloading it from [nuget.org](#). Add the `nuget.exe` file to a suitable folder, and add that folder to your PATH environment variable.

Create a class library project

You can use an existing .NET Class Library project for the code you want to package, or create one as follows:

1. In Visual Studio, select **File > New > Project**.
2. In the **Create a new project** window, select **C#, Windows, and Library** in the dropdown lists.
3. In the resulting list of project templates, select **Class Library** (with the description, *A project for creating a class library that targets .NET or .NET Standard*), and then select **Next**.
4. In the **Configure your new project** window, enter `AppLogger` for the **Project name**, and then select **Next**.
5. In the **Additional information** window, select an appropriate **Framework**, and then select **Create**.

If you're unsure which framework to select, the latest is a good choice, and can be easily changed later. For information about which framework to use, see [When to target .NET 5.0 or .NET 6.0 vs. .NET Standard](#).

6. To ensure the project was created properly, select **Build > Build Solution**. The DLL is found within the Debug folder (or Release if you build that configuration instead).
7. (Optional) For this quickstart, you don't need to write any additional code for the NuGet package because the template class library is sufficient to create a package. However, if you'd like some functional code for the package, include the following code:

```
C#  
  
namespace AppLogger  
{  
    public class Logger  
    {
```

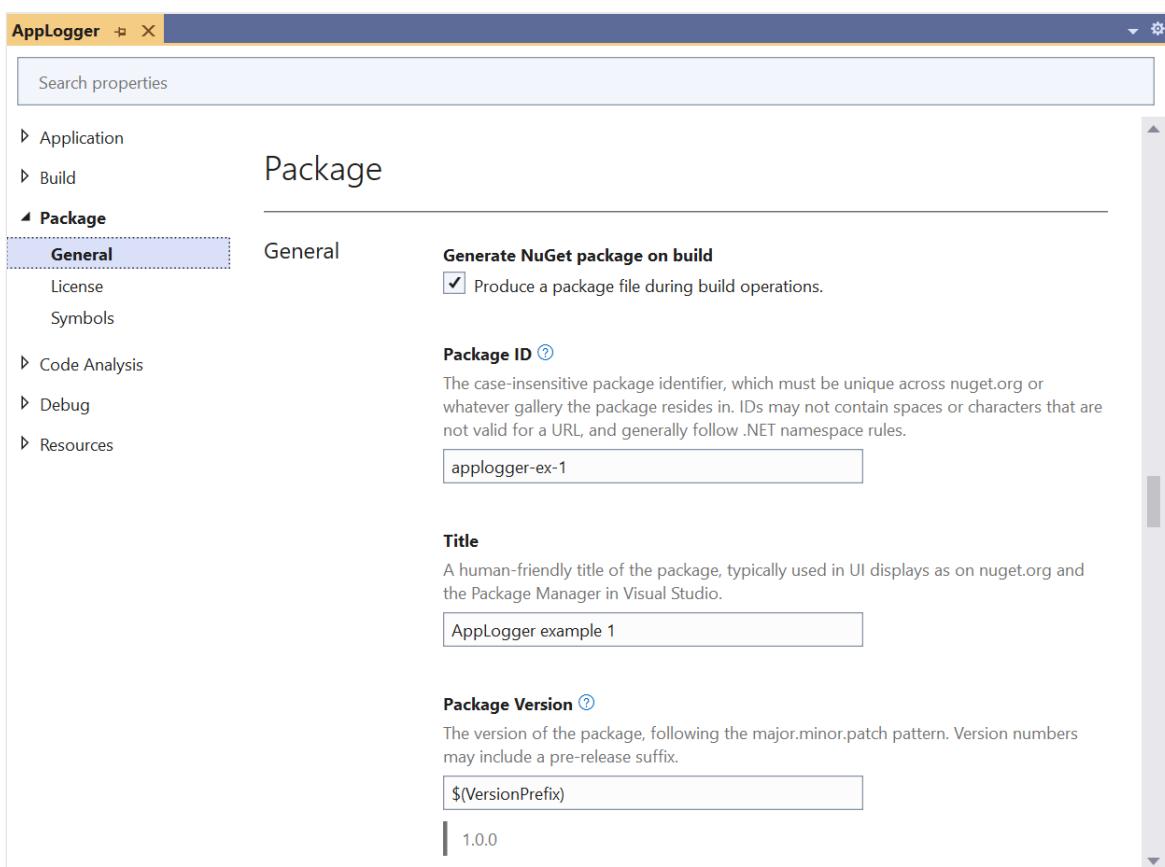
```
public void Log(string text)
{
    Console.WriteLine(text);
}
}
```

Configure package properties

After you've created your project, you can configure the NuGet package properties by following these steps:

1. Select your project in **Solution Explorer**, and then select **Project > <project name> Properties**, where <project name> is the name of your project.
2. Expand the **Package** node, and then select **General**.

The **Package** node appears only for SDK-style projects in Visual Studio. If you're targeting a non-SDK style project (typically .NET Framework), either [migrate the project](#), or see [Create and publish a .NET Framework package](#) for step-by-step instructions.



3. For packages built for public consumption, pay special attention to the **Tags** property, as tags help others find your package and understand what it does.

4. Give your package a unique **Package ID** and fill out any other desired properties.

For a table that shows how MSBuild properties (SDK-style projects) map to `.nuspec` file properties, see [pack targets](#). For a description of `.nuspec` file properties, see the [.nuspec file reference](#). All of these properties go into the `.nuspec` manifest that Visual Studio creates for the project.

 **Important**

You must give the package an identifier that's unique across nuget.org or whatever host you're using. Otherwise, an error occurs. For this quickstart we recommend including *Sample* or *Test* in the name because the publishing step makes the package publicly visible.

5. (Optional) To see the properties directly in the `AppLogger.csproj` project file, select **Project > Edit Project File**.

The `AppLogger.csproj` tab loads.

This option is available starting in Visual Studio 2017 for projects that use the SDK-style attribute. For earlier Visual Studio versions, you must select **Project > Unload Project** before you can edit the project file.

Run the pack command

To create a NuGet package from your project, follow these steps:

1. Select **Build > Configuration Manager**, and then set the **Active solution configuration** to **Release**.
 2. Select the AppLogger project in **Solution Explorer**, then select **Pack**.
- Visual Studio builds the project and creates the `.nupkg` file.
3. Examine the **Output** window for details, which contains the path to the package file. In this example, the built assembly is in `bin\Release\net6.0` as befits a .NET 6.0 target:

Output

```
1>----- Build started: Project: AppLogger, Configuration: Release Any CPU -----
1>AppLogger ->
d:\proj\AppLogger\AppLogger\bin\Release\net6.0\AppLogger.dll
1>Successfully created package
```

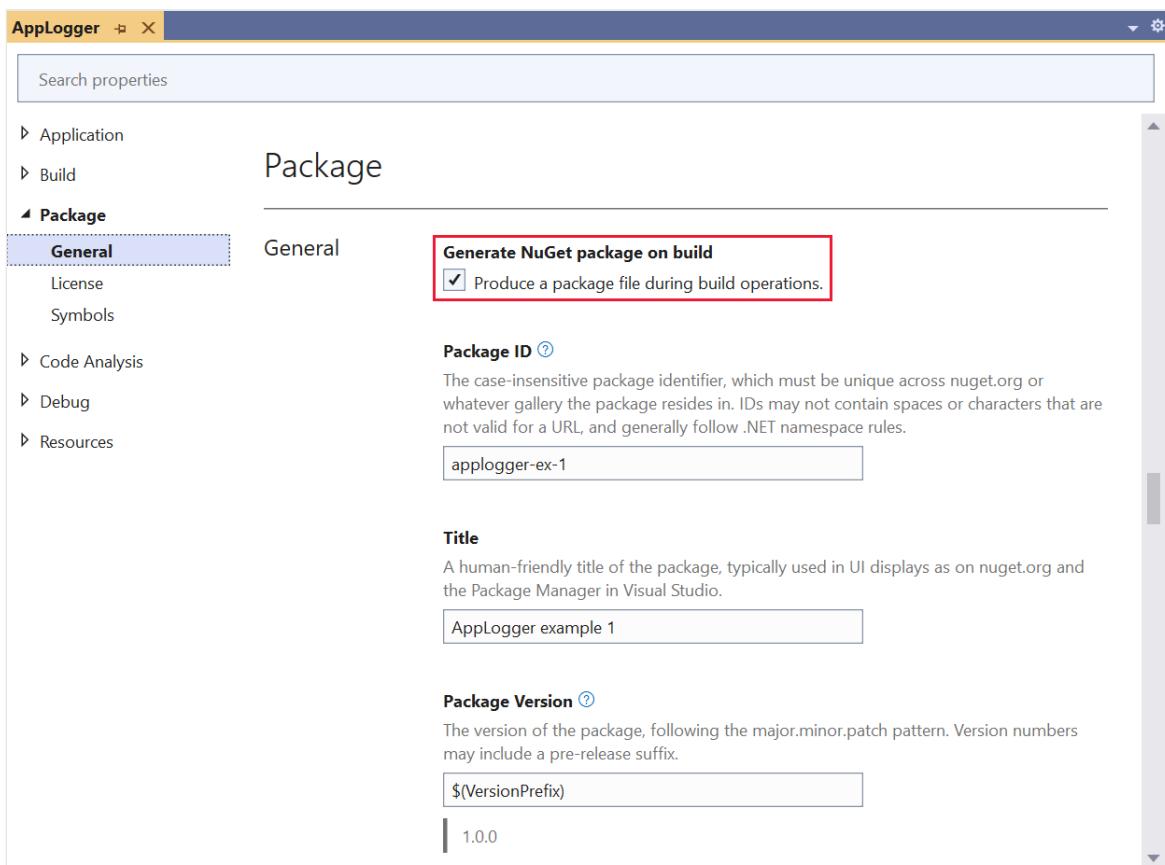
```
'd:\proj\AppLogger\AppLogger\bin\Release\AppLogger.1.0.0.nupkg'.
=====
Build: 1 succeeded, 0 failed, 0 up-to-date, 0 skipped
=====
```

4. If you don't see the **Pack** command on the menu, your project is probably not an SDK-style project, and you need to use the NuGet CLI. Either [migrate the project](#) and use .NET CLI, or see [Create and publish a .NET Framework package](#) for step-by-step instructions.

(Optional) Generate package on build

You can configure Visual Studio to automatically generate the NuGet package when you build the project:

1. Select your project in **Solution Explorer**, and then select **Project > <project name> Properties**, where <project name> is the name of your project (AppLogger in this case).
2. Expand the **Package** node, select **General**, and then select **Generate NuGet package on build**.



ⓘ Note

When you automatically generate the package, the extra time to pack increases the overall build time for your project.

(Optional) Pack with MSBuild

As an alternative to using the **Pack** menu command, NuGet 4.x+ and MSBuild 15.1+ supports a `pack` target when the project contains the necessary package data:

1. With your project open in **Solution Explorer**, open a command prompt by selecting **Tools > Command Line > Developer Command Prompt**.

The command prompt opens in your project directory.

2. Run the following command: `msbuild -t:pack`.

For more information, see [Create a package using MSBuild](#).

Publish the package

After you've created a `.nupkg` file, publish it to nuget.org by using either the .NET CLI or the NuGet CLI, along with an API key acquired from nuget.org.

Note

- Nuget.org scans all uploaded packages for viruses and rejects the packages if it finds any viruses. Nuget.org also scans all existing listed packages periodically.
- Packages you publish to nuget.org are publicly visible to other developers unless you unlist them. To host packages privately, see [Host your own NuGet feeds](#).

Acquire your API key

Before you publish your NuGet package, create an API key:

1. [Sign into your nuget.org account](#) or [create an account](#) if you don't have one already.
2. Select your user name at upper right, and then select **API Keys**.

3. Select **Create**, and provide a name for your key.

4. Under **Select Scopes**, select **Push**.

5. Under **Select Packages > Glob Pattern**, enter *****.

6. Select **Create**.

7. Select **Copy** to copy the new key.

⚠ Your API key has been regenerated. Make sure to copy your new API key now using the **Copy** button below. You will not be able to do so again.

 test-key1

⌚ Expires in a year | ⚡ Push new packages and package versions

Package owner: NuGet-test
Glob pattern: *

Copy   

Important

- Always keep your API key a secret. The API key is like a password that allows anyone to manage packages on your behalf. Delete or regenerate your API key if it's accidentally revealed.
- Save your key in a secure location, because you can't copy the key again later. If you return to the API key page, you need to regenerate the key to copy it. You can also remove the API key if you no longer want to push packages.

Scoping lets you create separate API keys for different purposes. Each key has an expiration timeframe, and you can scope the key to specific packages or glob patterns. You also scope each key to specific operations: Push new packages and package versions, push only new package versions, or unlist.

Through scoping, you can create API keys for different people who manage packages for your organization so they have only the permissions they need.

For more information, see [scoped API keys](#).

Publish with the .NET CLI or NuGet CLI

Each of the following CLI tools allows you to push a package to the server and publish it. Select the tab for your CLI tool, either .NET CLI or NuGet CLI.

.NET CLI

Using the .NET CLI (`dotnet.exe`) is the recommended alternative to using the NuGet CLI.

From the folder that contains the `.nupkg` file, run the following command. Specify your `.nupkg` filename, and replace the key value with your API key.

.NET CLI

```
dotnet nuget push Contoso.08.28.22.001.Test.1.0.0.nupkg --api-key  
qz2jga8pl3dvn2akksyquwcs9ygggg4exypy3bhxy6w6x6 --source  
https://api.nuget.org/v3/index.json
```

The output shows the results of the publishing process:

Output

```
Pushing Contoso.08.28.22.001.Test.1.0.0.nupkg to  
'https://www.nuget.org/api/v2/package'...  
PUT https://www.nuget.org/api/v2/package/  
warn : All published packages should have license information specified.  
Learn more: https://aka.ms/nuget/authoring-best-practices#licensing.  
Created https://www.nuget.org/api/v2/package/ 1221ms  
Your package was pushed.
```

For more information, see [dotnet nuget push](#).

ⓘ Note

If you want to avoid your test package being live on nuget.org, you can push to the nuget.org test site at <https://int.nugettest.org>. Note that packages uploaded to int.nugettest.org might not be preserved.

Publish errors

Errors from the `push` command typically indicate the problem. For example, you might have forgotten to update the version number in your project, so you're trying to publish a package that already exists.

You also see errors if your API key is invalid or expired, or if you try to publish a package using an identifier that already exists on the host. Suppose, for example, the identifier `AppLogger-test` already exists on nuget.org. If you try to publish a package with that identifier, the `push` command gives the following error:

Output

```
Response status code does not indicate success: 403 (The specified API key  
is invalid,  
has expired, or does not have permission to access the specified package.).
```

If you get this error, check that you're using a valid API key that hasn't expired. If you are, the error indicates the package identifier already exists on the host. To fix the error, change the package identifier to be unique, rebuild the project, recreate the `.nupkg` file, and retry the `push` command.

Manage the published package

When your package successfully publishes, you receive a confirmation email. To see the package you just published, on [nuget.org](#), select your user name at upper right, and then select **Manage Packages**.

ⓘ Note

It might take awhile for your package to be indexed and appear in search results where others can find it. During that time, your package appears under **Unlisted Packages**, and the package page shows the following message:

⚠ This package has not been published yet. It will appear in search results and will be available for install/restore after both validation and indexing are complete. Package validation and indexing may take up to an hour. [Read more](#).

You've now published a NuGet package to nuget.org that other developers can use in their projects.

If you've created a package that isn't useful (such as this sample package that was created with an empty class library), or you decide you don't want the package to be visible, you can *unlist* the package to hide it from search results:

1. After the package appears under **Published Packages** on the **Manage Packages** page, select the pencil icon next to the package listing.

| Published Packages | | | | | 1 package / 0 downloads |
|---|--------|---------------------------|-----------|----------------|---|
| Package ID | Owners | Signing Owner | Downloads | Latest Version | |
|  Contoso.08.28.22.001.Test | Test | username (0 certificates) | 0 | 1.0.0 |  |

2. On the next page, select **Listing**, deselect the **List in search results** checkbox, and then select **Save**.

✓ Listing

Select version

1.0.0 (Latest) ▾

List or unlist version

ⓘ You can control how your packages are listed using the checkbox below. As per [policy](#), permanent deletion is not supported as it would break every project depending on the availability of the package. For more assistance, [Contact Support](#).

List in search results.

Unlisted packages cannot be installed directly and do not show up in search results.

Save

The package now appears under **Unlisted Packages** in **Manage Packages** and no longer appears in search results.

ⓘ Note

To avoid your test package being live on nuget.org, you can push to the nuget.org test site at <https://int.nugettest.org>. Note that packages uploaded to int.nugettest.org might not be preserved.

Add a readme or another file

To directly specify files to include in the package, edit the project file and add the `content` property:

XML

```
<ItemGroup>
  <Content Include="readme.txt">
    <Pack>true</Pack>
    <PackagePath>\</PackagePath>
  </Content>
</ItemGroup>
```

In this example, the `Include` property specifies a file named `readme.txt` in the project root. Visual Studio displays the contents of that file as plain text immediately after it installs the package. Readme files aren't displayed for packages installed as dependencies. For example, here's the readme for the `HtmlAgilityPack` package:

Output

```
1 -----
2 ----- Html Agility Pack Nuget Readme -----
3 -----
4
5 ----Silverlight 4 and Windows Phone 7.1+ projects-----
6 To use XPATH features: System.Xml.XPath.dll from the 3 Silverlight 4 SDK
must be referenced.
7 This is normally found at
8 %ProgramFiles(x86)%\Microsoft SDKs\Microsoft
SDKs\Silverlight\v4.0\Libraries\Client
9 or
10 %ProgramFiles%\Microsoft SDKs\Microsoft
SDKs\Silverlight\v4.0\Libraries\Client
11
12 ----Silverlight 5 projects-----
13 To use XPATH features: System.Xml.XPath.dll from the Silverlight 5 SDK
must be referenced.
14 This is normally found at
15 %ProgramFiles(x86)%\Microsoft SDKs\Microsoft
SDKs\Silverlight\v5.0\Libraries\Client
16 or
17 %ProgramFiles%\Microsoft SDKs\Microsoft
SDKs\Silverlight\v5.0\Libraries\Client
```

Note

If you only add `readme.txt` at the project root without including it in the `content` property of the project file, it won't be included in the package.

Related video

<https://learn.microsoft.com/shows/NuGet-101/Create-and-Publish-a-NuGet-Package-with-Visual-Studio-4-of-5/player>

Find more NuGet videos on [Channel 9](#) and [YouTube](#).

Congratulations on creating a NuGet package by using a Visual Studio .NET class library. Advance to the next article to learn how to create a NuGet package with the Visual Studio .NET Framework.

Create a package using the NuGet CLI

To explore more that NuGet has to offer, see the following articles:

- [Create a NuGet package](#)
- [Publish a package](#)
- [Build a prerelease package](#)
- [Support multiple .NET Framework versions](#)
- [Package versioning](#)
- [Create localized NuGet packages](#)
- [Porting to .NET Core from .NET Framework](#)

Tutorial: Create a .NET console application using Visual Studio Code

Article • 09/30/2023

This tutorial shows how to create and run a .NET console application by using Visual Studio Code and the .NET CLI. Project tasks, such as creating, compiling, and running a project are done by using the .NET CLI. You can follow this tutorial with a different code editor and run commands in a terminal if you prefer.

Prerequisites

- [Visual Studio Code](#) with the [C# extension](#) installed.

If you have the [C# Dev Kit extension](#) installed, uninstall or disable it. It isn't used by this tutorial series.

For information about how to install extensions on Visual Studio Code, see [VS Code Extension Marketplace](#).

- The [.NET 8 SDK](#).

Create the app

Create a .NET console app project named "HelloWorld".

1. Start Visual Studio Code.
2. Select **File > Open Folder** (**File > Open...** on macOS) from the main menu.
3. In the **Open Folder** dialog, create a *HelloWorld* folder and select it. Then click **Select Folder (Open** on macOS).

The folder name becomes the project name and the namespace name by default. You'll add code later in the tutorial that assumes the project namespace is `HelloWorld`.

4. In the **Do you trust the authors of the files in this folder?** dialog, select **Yes, I trust the authors**. You can trust the authors because this folder only has files generated by .NET and added or modified by you.

5. Open the **Terminal** in Visual Studio Code by selecting **View > Terminal** from the main menu.

The **Terminal** opens with the command prompt in the *HelloWorld* folder.

6. In the **Terminal**, enter the following command:

```
.NET CLI  
dotnet new console --framework net8.0 --use-program-main
```

Open the *Program.cs* file to see the simple application created by the template:

```
C#  
  
namespace HelloWorld;  
  
class Program  
{  
    static void Main(string[] args)  
    {  
        Console.WriteLine("Hello, World!");  
    }  
}
```

The first time you open a *.cs* file, Visual Studio Code prompts you to add assets to build and debug your app. Select **Yes**, and Visual Studio Code creates a *.vscode* folder with *launch.json* and *tasks.json* files.

ⓘ Note

If you don't get the prompt, or if you accidentally dismiss it without selecting **Yes**, do the following steps to create *launch.json* and *tasks.json*:

- Select **Run > Add Configuration** from the menu.
- Select **.NET 5+ and .NET Core** at the **Select environment** prompt.

The code defines a class, `Program`, with a single method, `Main`, that takes a `String` array as an argument. `Main` is the application entry point, the method that's called automatically by the runtime when it launches the application. Any command-line arguments supplied when the application is launched are available in the `args` array. The code in `Main` calls the `Console.WriteLine(String)` method to display a message in the console window.

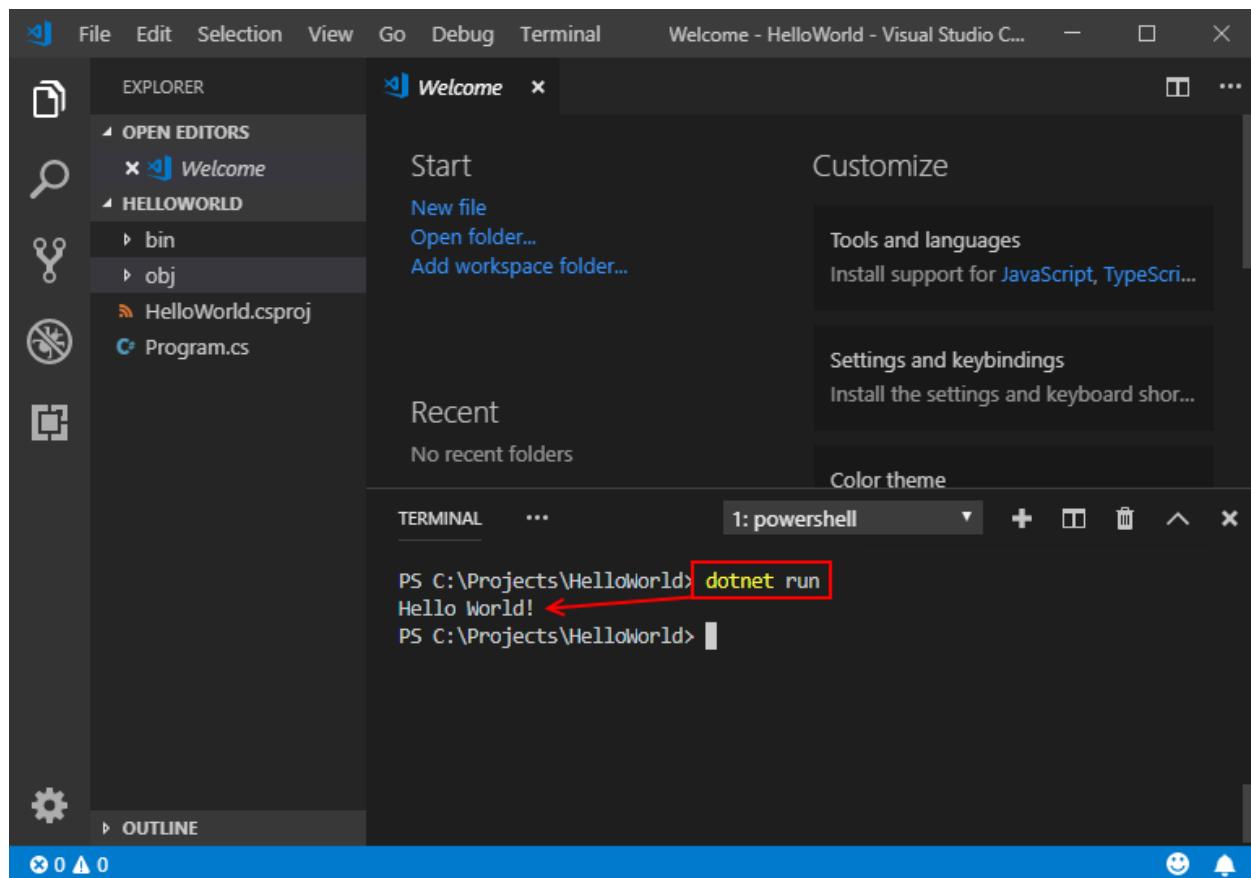
C# has a feature named [top-level statements](#) that lets you omit the `Program` class and the `Main` method. This tutorial doesn't use this feature. Whether you use it in your programs is a matter of style preference. In the `dotnet new` command that created the project, the `--use-program-main` option prevented top-level statements from being used.

Run the app

Run the following command in the Terminal:

```
.NET CLI  
dotnet run
```

The program displays "Hello, World!" and ends.



Enhance the app

Enhance the application to prompt the user for their name and display it along with the date and time.

1. Open `Program.cs`.

2. Replace the contents of the `Main` method in `Program.cs`, which is the line that calls `Console.WriteLine`, with the following code:

C#

```
Console.WriteLine("What is your name?");
var name = Console.ReadLine();
var currentDate = DateTime.Now;
Console.WriteLine(${Environment.NewLine}Hello, {name}, on
{currentDate:d} at {currentDate:t}!");
Console.Write(${Environment.NewLine}Press any key to exit...");
Console.ReadKey(true);
```

This code displays a prompt in the console window and waits until the user enters a string followed by the `Enter` key. It stores this string in a variable named `name`. It also retrieves the value of the `DateTime.Now` property, which contains the current local time, and assigns it to a variable named `currentDate`. And it displays these values in the console window. Finally, it displays a prompt in the console window and calls the `Console.ReadKey(Boolean)` method to wait for user input.

`NewLine` is a platform-independent and language-independent way to represent a line break. It's the same as `\n` in C#.

The dollar sign (\$) in front of a string lets you put expressions such as variable names in curly braces in the string. The expression value is inserted into the string in place of the expression. This syntax is referred to as [interpolated strings](#).

3. Save your changes.

 **Important**

In Visual Studio Code, you have to explicitly save changes. Unlike Visual Studio, file changes are not automatically saved when you build and run an app.

4. Run the program again:

.NET CLI

```
dotnet run
```

5. Respond to the prompt by entering a name and pressing the `Enter` key.

```
1  using System;
2
3  namespace HelloWorld
4  {
5      class Program
6      {
7          static void Main(string[] args)
8          {
9              Console.WriteLine("What is your name?");
10             var name = Console.ReadLine();
11             var currentDate = DateTime.Now;
12             Console.WriteLine($"{Environment.NewLine}Hello, {name},");
13             Console.WriteLine($"{Environment.NewLine}Press any key to exit...");
14             Console.ReadKey(true);
15         }
16     }
17 }
```

C:\Users\thrak\source\repos\ConsoleApp5> dotnet run
What is your name?
Nancy
Hello, Nancy, on 4/27/2021 at 8:52 AM!
Press any key to exit...

6. Press any key to exit the program.

Additional resources

- [Setting up Visual Studio Code](#)

Next steps

In this tutorial, you created a .NET console application. In the next tutorial, you debug the app.

[Debug a .NET console application using Visual Studio Code](#)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



.NET feedback

The .NET documentation is open source. Provide feedback here.

[Open a documentation issue](#)

[Provide product feedback](#)

Tutorial: Debug a .NET console application using Visual Studio Code

Article • 09/07/2023

This tutorial introduces the debugging tools available in Visual Studio Code for working with .NET apps.

Prerequisites

- This tutorial works with the console app that you create in [Create a .NET console application using Visual Studio Code](#).

Use Debug build configuration

Debug and *Release* are .NET's built-in build configurations. You use the *Debug* build configuration for debugging and the *Release* configuration for the final release distribution.

In the *Debug* configuration, a program compiles with full symbolic debug information and no optimization. Optimization complicates debugging, because the relationship between source code and generated instructions is more complex. The *release* configuration of a program has no symbolic debug information and is fully optimized.

By default, Visual Studio Code launch settings use the *Debug* build configuration, so you don't need to change it before debugging.

1. Start Visual Studio Code.
2. Open the folder of the project that you created in [Create a .NET console application using Visual Studio Code](#).

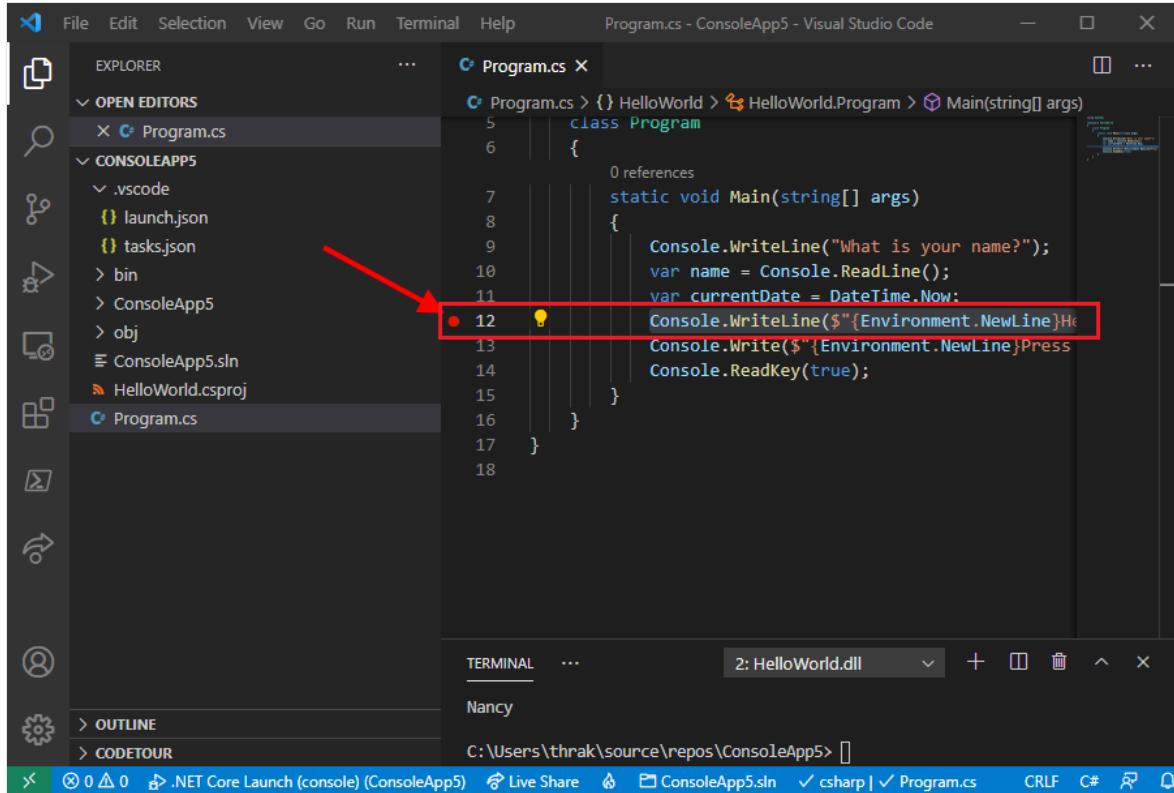
Set a breakpoint

A *breakpoint* temporarily interrupts the execution of the application before the line with the breakpoint is run.

1. Open the *Program.cs* file.
2. Set a *breakpoint* on the line that displays the name, date, and time, by clicking in the left margin of the code window. The left margin is to the left of the line

numbers. Other ways to set a breakpoint are by pressing **F9** or choosing **Run > Toggle Breakpoint** from the menu while the line of code is selected.

Visual Studio Code indicates the line on which the breakpoint is set by displaying a red dot in the left margin.



Set up for terminal input

The breakpoint is located after a `Console.ReadLine` method call. The **Debug Console** doesn't accept terminal input for a running program. To handle terminal input while debugging, you can use the integrated terminal (one of the Visual Studio Code windows) or an external terminal. For this tutorial, you use the integrated terminal.

1. The project folder contains a `.vscode` folder. Open the `launch.json` file that's in the `.vscode` folder.
2. In `launch.json`, change the `console` setting from `internalConsole` to `integratedTerminal`:

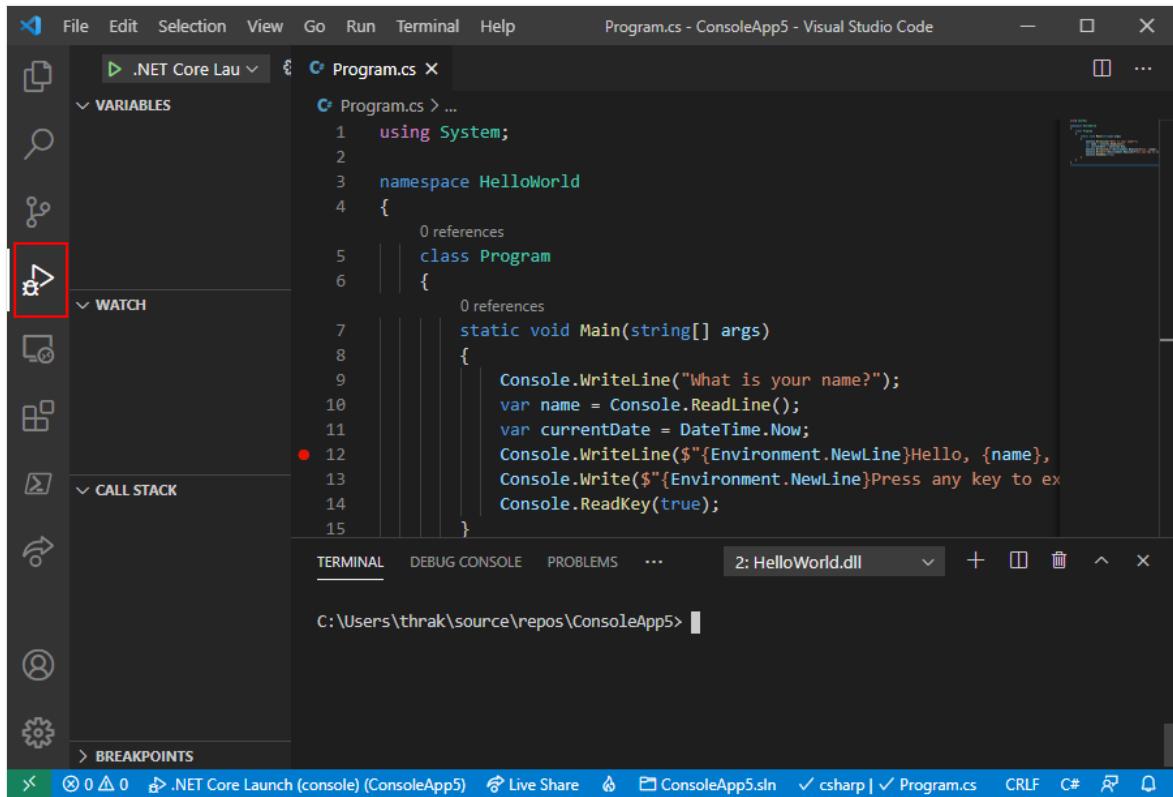
JSON

```
"console": "integratedTerminal",
```

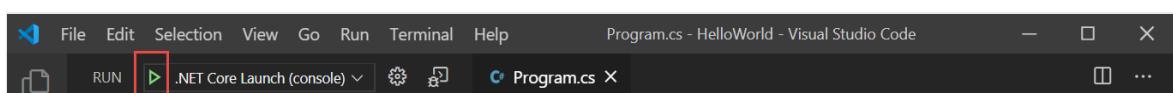
3. Save your changes.

Start debugging

1. Open the Debug view by selecting the Debugging icon on the left side menu.



2. Select the green arrow at the top of the pane, next to .NET Core Launch (console).
Other ways to start the program in debugging mode are by pressing **F5** or choosing Run > Start Debugging from the menu.



3. Select the Terminal tab to see the "What is your name?" prompt that the program displays before waiting for a response.

```
1  using System;
2
3  namespace HelloWorld
4  {
5      class Program
6      {
7          static void Main(string[] args)
8          {
9              Console.WriteLine("What is your name?");
10             var name = Console.ReadLine();
11             var currentDate = DateTime.Now;
12             Console.WriteLine($"{Environment.NewLine}Hello, {name},");
13             Console.WriteLine($"{Environment.NewLine}Press any key to exit.");
14             Console.ReadKey(true);
15     }
16 }
```

4. Enter a string in the **Terminal** window in response to the prompt for a name, and then press **Enter**.

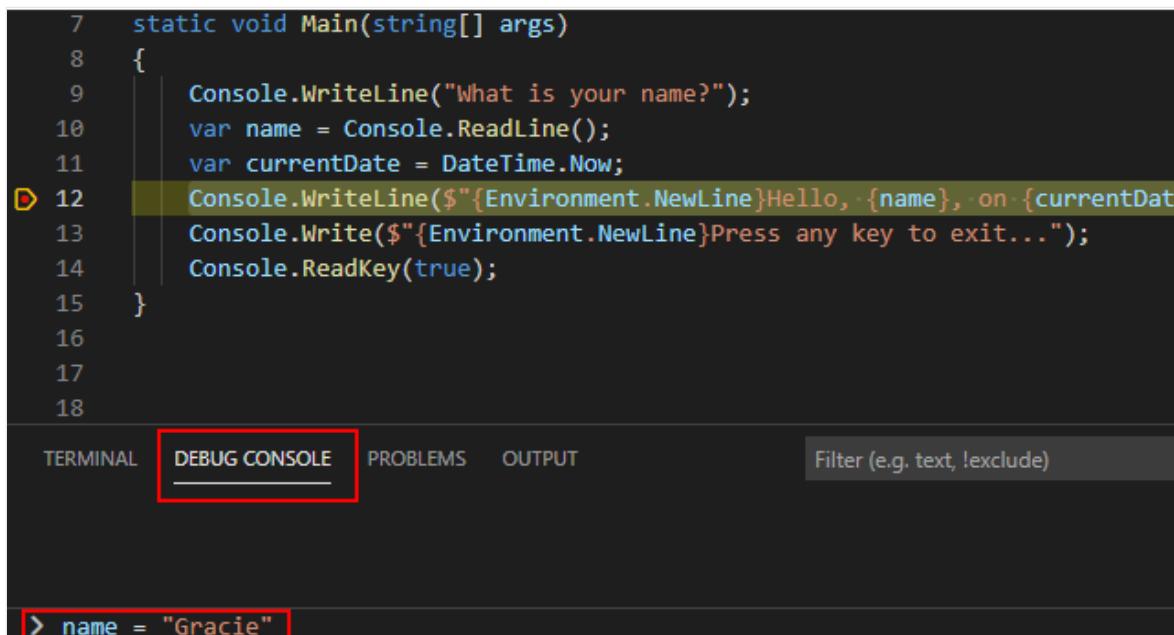
Program execution stops when it reaches the breakpoint and before the `Console.WriteLine` method runs. The **Locals** section of the **Variables** window displays the values of variables that are defined in the currently running method.

```
1  using System;
2
3  namespace HelloWorld
4  {
5      class Program
6      {
7          static void Main(string[] args)
8          {
9              Console.WriteLine("What is your name?");
10             var name = Console.ReadLine();
11             var currentDate = DateTime.Now;
12             Console.WriteLine($"{Environment.NewLine}Hello, {name},");
13             Console.WriteLine($"{Environment.NewLine}Press any key to exit.");
14             Console.ReadKey(true);
15     }
16 }
```

Use the Debug Console

The **Debug Console** window lets you interact with the application you're debugging. You can change the value of variables to see how it affects your program.

1. Select the **Debug Console** tab.
2. Enter `name = "Gracie"` at the prompt at the bottom of the **Debug Console** window and press the **Enter** key.



A screenshot of the Visual Studio Code interface. The code editor shows a C# file with the following code:

```
7 static void Main(string[] args)
8 {
9     Console.WriteLine("What is your name?");
10    var name = Console.ReadLine();
11    var currentDate = DateTime.Now;
12    Console.WriteLine($"{Environment.NewLine}Hello, {name}, on {currentDate}");
13    Console.WriteLine($"{Environment.NewLine}Press any key to exit...");
14    Console.ReadKey(true);
15 }
```

The line `Console.WriteLine($"{Environment.NewLine}Hello, {name}, on {currentDate}");` is highlighted with a yellow background. Below the code editor, the tabs are labeled TERMINAL, DEBUG CONSOLE, PROBLEMS, and OUTPUT. The DEBUG CONSOLE tab is highlighted with a red border. At the bottom of the screen, there is a terminal window with the prompt `> name = "Gracie"`, which is also highlighted with a red border.

3. Enter `currentDate = DateTime.Parse("2019-11-16T17:25:00Z").ToUniversalTime()` at the bottom of the **Debug Console** window and press the **Enter** key.

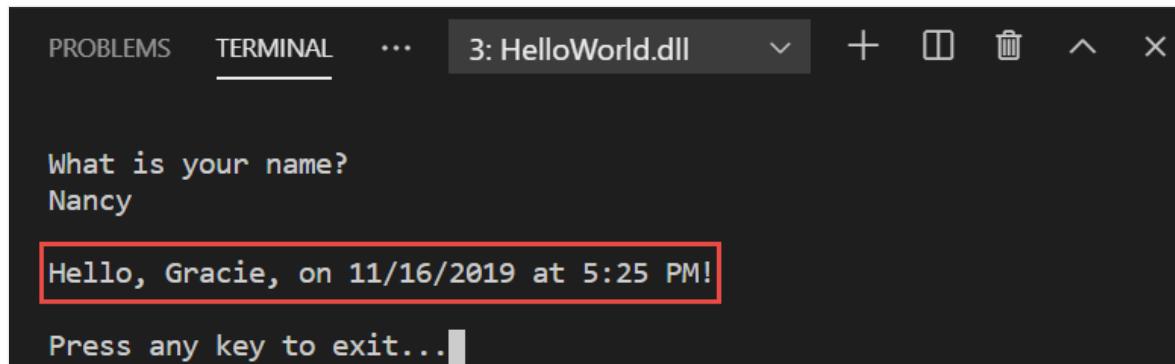
The **Variables** window displays the new values of the `name` and `currentDate` variables.

4. Continue program execution by selecting the **Continue** button in the toolbar.
Another way to continue is by pressing **F5**.



5. Select the **Terminal** tab again.

The values displayed in the console window correspond to the changes you made in the **Debug Console**.



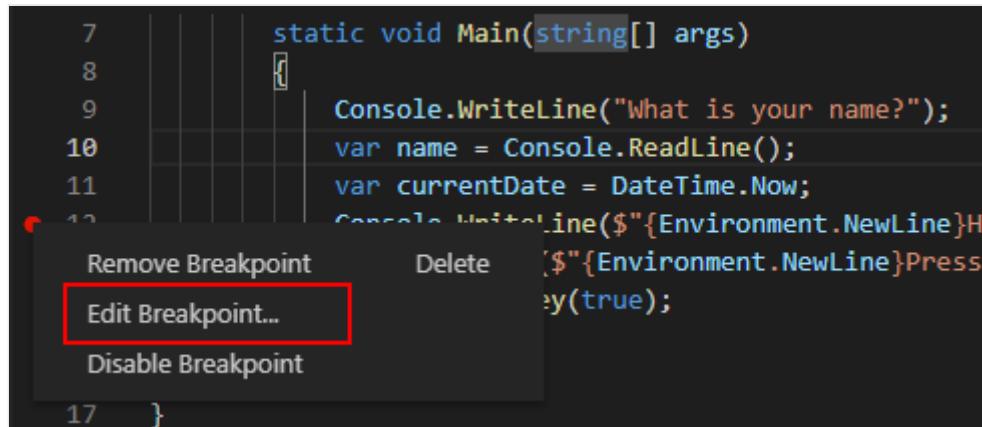
```
PROBLEMS TERMINAL ... 3: HelloWorld.dll + ⌂ ⌂ ⌂ X
What is your name?
Nancy
Hello, Gracie, on 11/16/2019 at 5:25 PM!
Press any key to exit...
```

6. Press any key to exit the application and stop debugging.

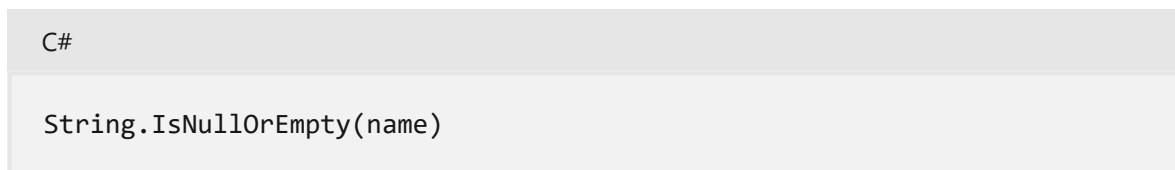
Set a conditional breakpoint

The program displays the string that the user enters. What happens if the user doesn't enter anything? You can test this with a useful debugging feature called a *conditional breakpoint*.

1. Right-click (ctrl-click on macOS) on the red dot that represents the breakpoint. In the context menu, select **Edit Breakpoint** to open a dialog that lets you enter a conditional expression.



2. Select **Expression** in the drop-down, enter the following conditional expression, and press **Enter**.



```
7     static void Main(string[] args)
8     {
9         Console.WriteLine("What is your name?");
10        var name = Console.ReadLine();
11        var currentDate = DateTime.Now;
12        Console.WriteLine($"{Environment.NewLine}Hello, {name}, on {currentD
13        Console.Write($"{Environment.NewLine}Press any key to exit...");
14        Console.ReadKey(true);
15    }
```

Each time the breakpoint is hit, the debugger calls the `String.IsNullOrEmpty(name)` method, and it breaks on this line only if the method call returns `true`.

Instead of a conditional expression, you can specify a *hit count*, which interrupts program execution before a statement is run a specified number of times. Another option is to specify a *filter condition*, which interrupts program execution based on such attributes as a thread identifier, process name, or thread name.

3. Start the program with debugging by pressing `F5`.
4. In the **Terminal** tab, press the `Enter` key when prompted to enter your name.

Because the condition you specified (`name` is either `null` or `String.Empty`) has been satisfied, program execution stops when it reaches the breakpoint and before the `Console.WriteLine` method runs.

The **Variables** window shows that the value of the `name` variable is `""`, or `String.Empty`.

5. Confirm the value is an empty string by entering the following statement at the **Debug Console** prompt and pressing `Enter`. The result is `true`.

```
C#
name == String.Empty
```

6. Select the **Continue** button on the toolbar to continue program execution.
7. Select the **Terminal** tab, and press any key to exit the program and stop debugging.
8. Clear the breakpoint by clicking on the dot in the left margin of the code window. Other ways to clear a breakpoint are by pressing `F9` or choosing **Run > Toggle Breakpoint** from the menu while the line of code is selected.

9. If you get a warning that the breakpoint condition will be lost, select **Remove Breakpoint**.

Step through a program

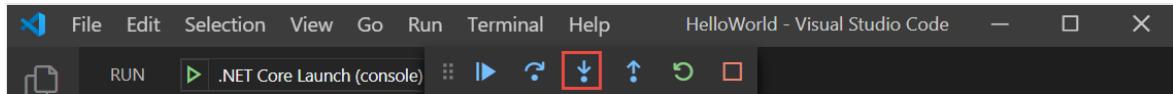
Visual Studio Code also allows you to step line by line through a program and monitor its execution. Ordinarily, you'd set a breakpoint and follow program flow through a small part of your program code. Since this program is small, you can step through the entire program.

1. Set a breakpoint on the opening curly brace of the `Main` method.
2. Press `F5` to start debugging.

Visual Studio Code highlights the breakpoint line.

At this point, the **Variables** window shows that the `args` array is empty, and `name` and `currentDate` have default values.

3. Select **Run > Step Into** or press `F11`.



Visual Studio Code highlights the next line.

4. Select **Run > Step Into** or press `F11`.

Visual Studio Code runs the `Console.WriteLine` for the name prompt and highlights the next line of execution. The next line is the `Console.ReadLine` for the `name`. The **Variables** window is unchanged, and the **Terminal** tab shows the "What is your name?" prompt.

5. Select **Run > Step Into** or press `F11`.

Visual Studio highlights the `name` variable assignment. The **Variables** window shows that `name` is still `null`.

6. Respond to the prompt by entering a string in the Terminal tab and pressing `Enter`.

The **Terminal** tab might not display the string you enter while you're entering it, but the `Console.ReadLine` method will capture your input.

7. Select **Run > Step Into** or press `F11`.

Visual Studio Code highlights the `currentDate` variable assignment. The **Variables** window shows the value returned by the call to the `Console.ReadLine` method. The **Terminal** tab displays the string you entered at the prompt.

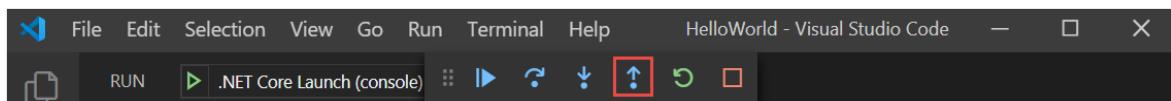
8. Select **Run > Step Into** or press `F11`.

The **Variables** window shows the value of the `currentDate` variable after the assignment from the `DateTime.Now` property.

9. Select **Run > Step Into** or press `F11`.

Visual Studio Code calls the `Console.WriteLine(String, Object, Object)` method. The console window displays the formatted string.

10. Select **Run > Step Out** or press `Shift + F11`.



11. Select the **Terminal** tab.

The terminal displays "Press any key to exit..."

12. Press any key to exit the program.

Use Release build configuration

Once you've tested the Debug version of your application, you should also compile and test the Release version. The Release version incorporates compiler optimizations that can affect the behavior of an application. For example, compiler optimizations that are designed to improve performance can create race conditions in multithreaded applications.

To build and test the Release version of your console application, open the **Terminal** and run the following command:

```
.NET CLI
dotnet run --configuration Release
```

Additional resources

- [Debugging in Visual Studio Code ↗](#)

Next steps

In this tutorial, you used Visual Studio Code debugging tools. In the next tutorial, you publish a deployable version of the app.

Publish a .NET console application using Visual Studio Code

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

Tutorial: Publish a .NET console application using Visual Studio Code

Article • 09/07/2023

This tutorial shows how to publish a console app so that other users can run it. Publishing creates the set of files that are needed to run an application. To deploy the files, copy them to the target machine.

The .NET CLI is used to publish the app, so you can follow this tutorial with a code editor other than Visual Studio Code if you prefer.

Prerequisites

- This tutorial works with the console app that you create in [Create a .NET console application using Visual Studio Code](#).

Publish the app

1. Start Visual Studio Code.
2. Open the *HelloWorld* project folder that you created in [Create a .NET console application using Visual Studio Code](#).
3. Choose **View > Terminal** from the main menu.

The terminal opens in the *HelloWorld* folder.

4. Run the following command:

```
.NET CLI  
dotnet publish --configuration Release
```

The default build configuration is *Debug*, so this command specifies the *Release* build configuration. The output from the *Release* build configuration has minimal symbolic debug information and is fully optimized.

The command output is similar to the following example:

```
Output
```

```
Microsoft (R) Build Engine version 17.8.0+b89cb5fde for .NET
Copyright (C) Microsoft Corporation. All rights reserved.

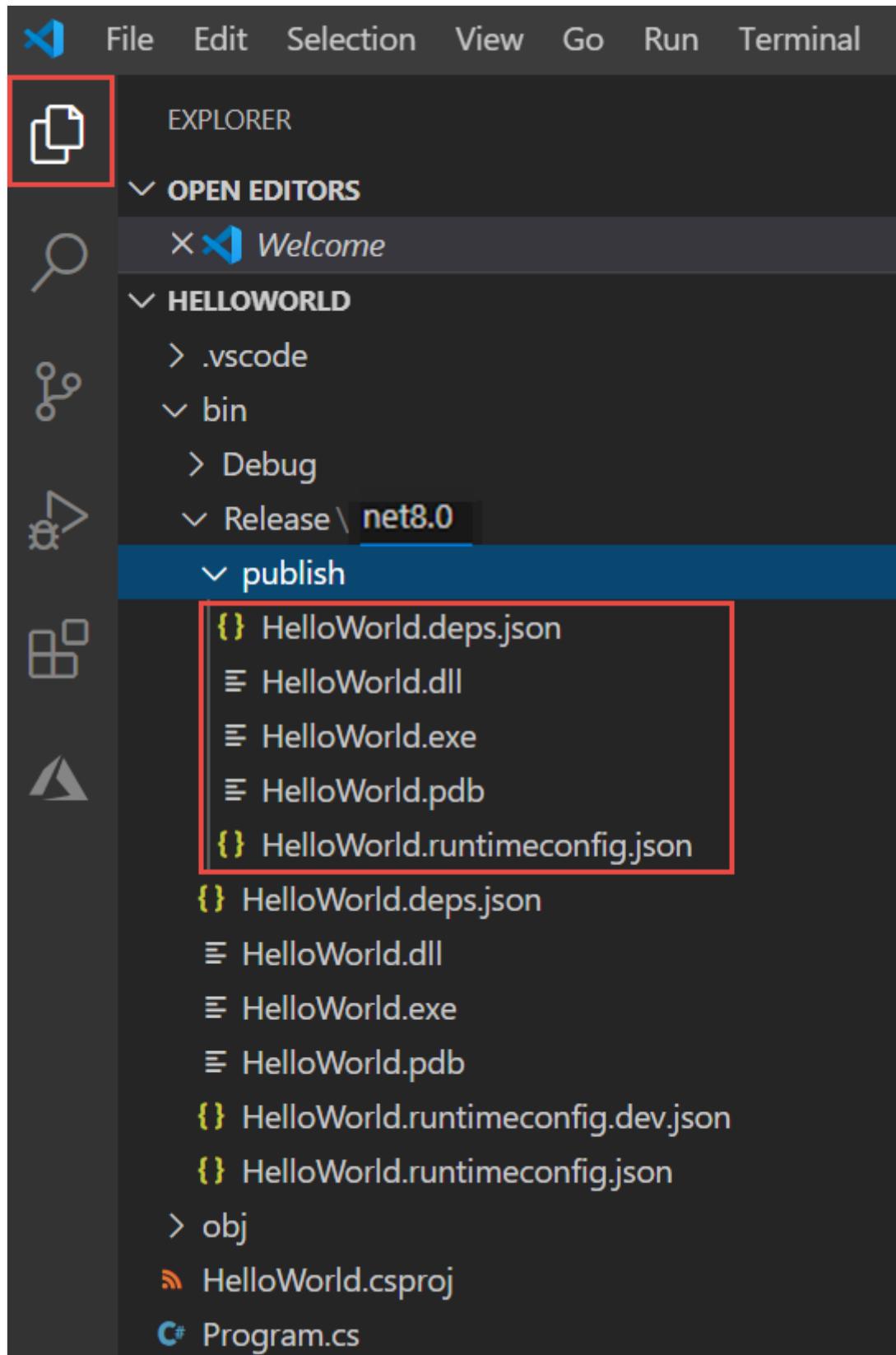
Determining projects to restore...
All projects are up-to-date for restore.
HelloWorld ->
C:\Projects\HelloWorld\bin\Release\net8.0\HelloWorld.dll
HelloWorld -> C:\Projects\HelloWorld\bin\Release\net8.0\publish\
```

Inspect the files

By default, the publishing process creates a framework-dependent deployment, which is a type of deployment where the published application runs on a machine that has the .NET runtime installed. To run the published app you can use the executable file or run the `dotnet HelloWorld.dll` command from a command prompt.

In the following steps, you'll look at the files created by the publish process.

1. Select the **Explorer** in the left navigation bar.
2. Expand *bin/Release/net7.0/publish*.



As the image shows, the published output includes the following files:

- *HelloWorld.deps.json*

This is the application's runtime dependencies file. It defines the .NET components and the libraries (including the dynamic link library that contains your application) needed to run the app. For more information, see [Runtime configuration files](#).

- *HelloWorld.dll*

This is the [framework-dependent deployment](#) version of the application. To run this dynamic link library, enter `dotnet HelloWorld.dll` at a command prompt. This method of running the app works on any platform that has the .NET runtime installed.

- *HelloWorld.exe* (*HelloWorld* on Linux, not created on macOS.)

This is the [framework-dependent executable](#) version of the application. The file is operating-system-specific.

- *HelloWorld.pdb* (optional for deployment)

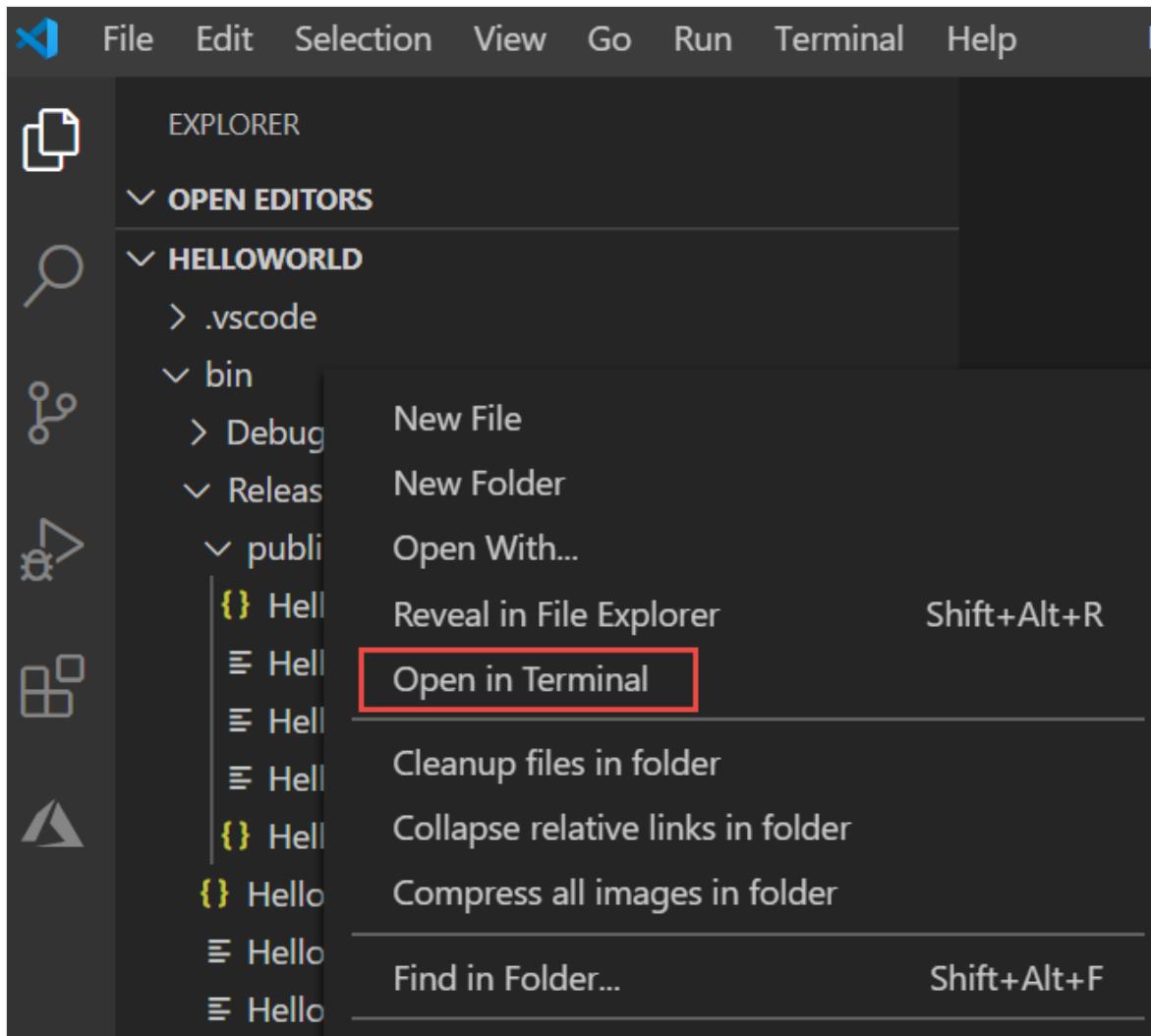
This is the debug symbols file. You aren't required to deploy this file along with your application, although you should save it in the event that you need to debug the published version of your application.

- *HelloWorld.runtimeconfig.json*

This is the application's runtime configuration file. It identifies the version of .NET that your application was built to run on. You can also add configuration options to it. For more information, see [.NET runtime configuration settings](#).

Run the published app

1. In Explorer, right-click the *publish* folder (`Ctrl`-click on macOS), and select **Open in Integrated Terminal**.



2. On Windows or Linux, run the app by using the executable.

a. On Windows, enter `.\HelloWorld.exe` and press `Enter`.

b. On Linux, enter `./HelloWorld` and press `Enter`.

c. Enter a name in response to the prompt, and press any key to exit.

3. On any platform, run the app by using the `dotnet` command:

a. Enter `dotnet HelloWorld.dll` and press `Enter`.

b. Enter a name in response to the prompt, and press any key to exit.

Additional resources

- [.NET application deployment](#)
- [Publish .NET apps with the .NET CLI](#)
- [dotnet publish](#)
- [Use the .NET SDK in continuous integration \(CI\) environments](#)

Next steps

In this tutorial, you published a console app. In the next tutorial, you create a class library.

[Create a .NET class library using Visual Studio Code](#)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

.NET is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

Tutorial: Create a .NET class library using Visual Studio Code

Article • 09/07/2023

In this tutorial, you create a simple utility library that contains a single string-handling method.

A *class library* defines types and methods that are called by an application. If the library targets .NET Standard 2.0, it can be called by any .NET implementation (including .NET Framework) that supports .NET Standard 2.0. If the library targets .NET 8, it can be called by any application that targets .NET 8. This tutorial shows how to target .NET 8.

When you create a class library, you can distribute it as a third-party component or as a bundled component with one or more applications.

Prerequisites

- [Visual Studio Code](#) with the [C# extension](#) installed.

If you have the [C# Dev Kit extension](#) installed, uninstall or disable it. It isn't used by this tutorial series.

For information about how to install extensions on Visual Studio Code, see [VS Code Extension Marketplace](#).

- The [.NET 8 SDK](#).

Create a solution

Start by creating a blank solution to put the class library project in. A solution serves as a container for one or more projects. You'll add additional, related projects to the same solution.

1. Start Visual Studio Code.
2. Select **File > Open Folder (Open... on macOS)** from the main menu
3. In the **Open Folder** dialog, create a *ClassLibraryProjects* folder and click **Select Folder (Open on macOS)**.

4. Open the **Terminal** in Visual Studio Code by selecting **View > Terminal** from the main menu.

The **Terminal** opens with the command prompt in the *ClassLibraryProjects* folder.

5. In the **Terminal**, enter the following command:

```
.NET CLI  
dotnet new sln
```

The terminal output looks like the following example:

```
Output  
The template "Solution File" was created successfully.
```

Create a class library project

Add a new .NET class library project named "StringLibrary" to the solution.

1. In the terminal, run the following command to create the library project:

```
.NET CLI  
dotnet new classlib -o StringLibrary
```

The `-o` or `--output` command specifies the location to place the generated output.

The terminal output looks like the following example:

```
Output  
The template "Class library" was created successfully.  
Processing post-creation actions...  
Running 'dotnet restore' on StringLibrary\StringLibrary.csproj...  
Determining projects to restore...  
Restored  
C:\Projects\ClassLibraryProjects\StringLibrary\StringLibrary.csproj (in  
328 ms).  
Restore succeeded.
```

2. Run the following command to add the library project to the solution:

.NET CLI

```
dotnet sln add StringLibrary/StringLibrary.csproj
```

The terminal output looks like the following example:

Output

```
Project `StringLibrary\StringLibrary.csproj` added to the solution.
```

3. Check to make sure that the library targets .NET 8. In Explorer, open *StringLibrary/StringLibrary.csproj*.

The `TargetFramework` element shows that the project targets .NET 8.0.

XML

```
<Project Sdk="Microsoft.NET.Sdk">

    <PropertyGroup>
        <TargetFramework>net8.0</TargetFramework>
    </PropertyGroup>

</Project>
```

4. Open *Class1.cs* and replace the code with the following code.

C#

```
namespace UtilityLibraries;

public static class StringLibrary
{
    public static bool StartsWithUpper(this string? str)
    {
        if (string.IsNullOrWhiteSpace(str))
            return false;

        char ch = str[0];
        return char.IsUpper(ch);
    }
}
```

The class library, `UtilityLibraries.StringLibrary`, contains a method named `StartsWithUpper`. This method returns a `Boolean` value that indicates whether the current string instance begins with an uppercase character. The Unicode standard

distinguishes uppercase characters from lowercase characters. The [Char.IsUpper\(Char\)](#) method returns `true` if a character is uppercase.

`StartsWithUpper` is implemented as an [extension method](#) so that you can call it as if it were a member of the [String](#) class.

5. Save the file.
6. Run the following command to build the solution and verify that the project compiles without error.

```
.NET CLI
```

```
dotnet build
```

The terminal output looks like the following example:

```
Output
```

```
Microsoft (R) Build Engine version 17.8.0+b89cb5fde for .NET
Copyright (C) Microsoft Corporation. All rights reserved.

Determining projects to restore...
All projects are up-to-date for restore.

StringLibrary ->
C:\Projects\ClassLibraryProjects\StringLibrary\bin\Debug\net8.0\StringL
ibrary.dll
Build succeeded.

0 Warning(s)
0 Error(s)
Time Elapsed 00:00:02.78
```

Add a console app to the solution

Add a console application that uses the class library. The app will prompt the user to enter a string and report whether the string begins with an uppercase character.

1. In the terminal, run the following command to create the console app project:

```
.NET CLI
```

```
dotnet new console -o Showcase
```

The terminal output looks like the following example:

```
Output
```

```
The template "Console Application" was created successfully.  
Processing post-creation actions...  
Running 'dotnet restore' on Showcase\ShowCase.csproj...  
Determining projects to restore...  
Restored C:\Projects\ClassLibraryProjects>ShowCase>ShowCase.csproj  
(in 210 ms).  
Restore succeeded.
```

- Run the following command to add the console app project to the solution:

.NET CLI

```
dotnet sln add Showcase/ShowCase.csproj
```

The terminal output looks like the following example:

Output

```
Project `ShowCase>ShowCase.csproj` added to the solution.
```

- Open *ShowCase/Program.cs* and replace all of the code with the following code.

C#

```
using UtilityLibraries;

class Program
{
    static void Main(string[] args)
    {
        int row = 0;

        do
        {
            if (row == 0 || row >= 25)
                ResetConsole();

            string? input = Console.ReadLine();
            if (string.IsNullOrEmpty(input)) break;
            Console.WriteLine($"Input: {input}");
            Console.WriteLine("Begins with uppercase? " +
                $"{(input.StartsWithUpper() ? "Yes" : "No")}");
            Console.WriteLine();
            row += 4;
        } while (true);
        return;

        // Declare a ResetConsole local method
        void ResetConsole()
```

```
        {
            if (row > 0)
            {
                Console.WriteLine("Press any key to continue...");
                Console.ReadKey();
            }
            Console.Clear();
            Console.WriteLine($"{Environment.NewLine}Press <Enter> only
to exit; otherwise, enter a string and press <Enter>:
{Environment.NewLine}");
            row = 3;
        }
    }
}
```

The code uses the `row` variable to maintain a count of the number of rows of data written to the console window. Whenever it's greater than or equal to 25, the code clears the console window and displays a message to the user.

The program prompts the user to enter a string. It indicates whether the string starts with an uppercase character. If the user presses the `Enter` key without entering a string, the application ends, and the console window closes.

4. Save your changes.

Add a project reference

Initially, the new console app project doesn't have access to the class library. To allow it to call methods in the class library, create a project reference to the class library project.

1. Run the following command:

```
.NET CLI
dotnet add ShowCase>ShowCase.csproj reference
StringLibrary/StringLibrary.csproj
```

The terminal output looks like the following example:

```
Output
Reference `..\StringLibrary\StringLibrary.csproj` added to the project.
```

Run the app

1. Run the following command in the terminal:

.NET CLI

```
dotnet run --project ShowCase/ShowCase.csproj
```

2. Try out the program by entering strings and pressing `Enter`, then press `Enter` to exit.

The terminal output looks like the following example:

Output

```
Press <Enter> only to exit; otherwise, enter a string and press <Enter>:
```

```
A string that starts with an uppercase letter
```

```
Input: A string that starts with an uppercase letter
```

```
Begins with uppercase? : Yes
```

```
a string that starts with a lowercase letter
```

```
Input: a string that starts with a lowercase letter
```

```
Begins with uppercase? : No
```

Additional resources

- [Develop libraries with the .NET CLI](#)
- [.NET Standard versions and the platforms they support.](#)

Next steps

In this tutorial, you created a solution, added a library project, and added a console app project that uses the library. In the next tutorial, you add a unit test project to the solution.

[Test a .NET class library with .NET using Visual Studio Code](#)

 Collaborate with us on
GitHub

The source for this content can
be found on GitHub, where you

.NET

.NET feedback

The .NET documentation is open
source. Provide feedback here.

can also create and review issues and pull requests. For more information, see [our contributor guide](#).

 [Open a documentation issue](#)

 [Provide product feedback](#)

Tutorial: Test a .NET class library using Visual Studio Code

Article • 09/07/2023

This tutorial shows how to automate unit testing by adding a test project to a solution.

Prerequisites

- This tutorial works with the solution that you create in [Create a .NET class library using Visual Studio Code](#).

Create a unit test project

Unit tests provide automated software testing during your development and publishing. The testing framework that you use in this tutorial is MSTest. [MSTest](#) is one of three test frameworks you can choose from. The others are [xUnit](#) and [nUnit](#).

1. Start Visual Studio Code.
2. Open the `ClassLibraryProjects` solution you created in [Create a .NET class library using Visual Studio Code](#).
3. Create a unit test project named "StringLibraryTest".

```
.NET CLI  
dotnet new mstest -o StringLibraryTest
```

The project template creates a `UnitTest1.cs` file with the following code:

```
C#  
  
namespace StringLibraryTest;  
  
[TestClass]  
public class UnitTest1  
{  
    [TestMethod]  
    public void TestMethod1()  
    {  
    }  
}
```

The source code created by the unit test template does the following:

- It applies the `TestClassAttribute` attribute to the `UnitTest1` class.
- It applies the `TestMethodAttribute` attribute to define `TestMethod1`.
- It imports the `Microsoft.VisualStudio.TestTools.UnitTesting` namespace, which contains the types used for unit testing. The namespace is imported via a `global using` directive in `GlobalUsings.cs`.

Each method tagged with `[TestMethod]` in a test class tagged with `[TestClass]` is run automatically when the unit test is invoked.

4. Add the test project to the solution.

.NET CLI

```
dotnet sln add StringLibraryTest/StringLibraryTest.csproj
```

Add a project reference

For the test project to work with the `StringLibrary` class, add a reference in the `StringLibraryTest` project to the `StringLibrary` project.

1. Run the following command:

.NET CLI

```
dotnet add StringLibraryTest/StringLibraryTest.csproj reference  
StringLibrary/StringLibrary.csproj
```

Add and run unit test methods

When Visual Studio invokes a unit test, it runs each method that is marked with the `TestMethodAttribute` attribute in a class that is marked with the `TestClassAttribute` attribute. A test method ends when the first failure is found or when all tests contained in the method have succeeded.

The most common tests call members of the `Assert` class. Many assert methods include at least two parameters, one of which is the expected test result and the other of which is the actual test result. Some of the `Assert` class's most frequently called methods are shown in the following table:

| Assert methods | Function |
|-------------------------------|--|
| <code>Assert.AreEqual</code> | Verifies that two values or objects are equal. The assert fails if the values or objects aren't equal. |
| <code>Assert.AreSame</code> | Verifies that two object variables refer to the same object. The assert fails if the variables refer to different objects. |
| <code>Assert.IsFalse</code> | Verifies that a condition is <code>false</code> . The assert fails if the condition is <code>true</code> . |
| <code>Assert.IsNotNull</code> | Verifies that an object isn't <code>null</code> . The assert fails if the object is <code>null</code> . |

You can also use the `Assert.ThrowsException` method in a test method to indicate the type of exception it's expected to throw. The test fails if the specified exception isn't thrown.

In testing the `StringLibrary.StartsWithUpper` method, you want to provide a number of strings that begin with an uppercase character. You expect the method to return `true` in these cases, so you can call the `Assert.IsTrue` method. Similarly, you want to provide a number of strings that begin with something other than an uppercase character. You expect the method to return `false` in these cases, so you can call the `Assert.IsFalse` method.

Since your library method handles strings, you also want to make sure that it successfully handles an `empty string (String.Empty)` and a `null` string. An empty string is one that has no characters and whose `Length` is 0. A `null` string is one that hasn't been initialized. You can call `StartsWithUpper` directly as a static method and pass a single `String` argument. Or you can call `StartsWithUpper` as an extension method on a `string` variable assigned to `null`.

You'll define three methods, each of which calls an `Assert` method for each element in a string array. You'll call a method overload that lets you specify an error message to be displayed in case of test failure. The message identifies the string that caused the failure.

To create the test methods:

1. Open `StringLibraryTest/UnitTest1.cs` and replace all of the code with the following code.

C#

```
using Microsoft.VisualStudio.TestTools.UnitTesting;
using UtilityLibraries;

namespace StringLibraryTest;
```

```
[TestClass]
public class UnitTest1
{
    [TestMethod]
    public void TestStartsWithUpper()
    {
        // Tests that we expect to return true.
        string[] words = { "Alphabet", "Zebra", "ABC", "Αθήνα",
"Москва" };
        foreach (var word in words)
        {
            bool result = word.StartsWithUpper();
            Assert.IsTrue(result,
                string.Format("Expected for '{0}': true; Actual:
{1}",
                                word, result));
        }
    }

    [TestMethod]
    public void TestDoesNotStartWithUpper()
    {
        // Tests that we expect to return false.
        string[] words = { "alphabet", "zebra", "abc",
"αυτοκινητοβιομηχανία", "государство",
"1234", ".", ";", " " };
        foreach (var word in words)
        {
            bool result = word.StartsWithUpper();
            Assert.IsFalse(result,
                string.Format("Expected for '{0}': false; Actual:
{1}",
                                word, result));
        }
    }

    [TestMethod]
    public void DirectCallWithNullOrEmpty()
    {
        // Tests that we expect to return false.
        string?[] words = { string.Empty, null };
        foreach (var word in words)
        {
            bool result = StringLibrary.StartsWithUpper(word);
            Assert.IsFalse(result,
                string.Format("Expected for '{0}': false; Actual:
{1}",
                                word == null ? "<null>" : word,
result));
        }
    }
}
```

The test of uppercase characters in the `TestStartsWithUpper` method includes the Greek capital letter alpha (U+0391) and the Cyrillic capital letter EM (U+041C). The test of lowercase characters in the `TestDoesNotStartWithUpper` method includes the Greek small letter alpha (U+03B1) and the Cyrillic small letter Ghe (U+0433).

2. Save your changes.

3. Run the tests:

.NET CLI

```
dotnet test StringLibraryTest/StringLibraryTest.csproj
```

The terminal output shows that all tests passed.

Output

```
Starting test execution, please wait...
A total of 1 test files matched the specified pattern.

Passed! - Failed: 0, Passed: 3, Skipped: 0, Total: 3,
Duration: 3 ms - StringLibraryTest.dll (net8.0)
```

Handle test failures

If you're doing test-driven development (TDD), you write tests first and they fail the first time you run them. Then you add code to the app that makes the test succeed. For this tutorial, you created the test after writing the app code that it validates, so you haven't seen the test fail. To validate that a test fails when you expect it to fail, add an invalid value to the test input.

1. Modify the `words` array in the `TestDoesNotStartWithUpper` method to include the string "Error".

C#

```
string[] words = { "alphabet", "Error", "zebra", "abc",
    "автокинητοβιομηχανία", "государство",
    "1234", ".", ";" };
```

2. Run the tests:

.NET CLI

```
dotnet test StringLibraryTest/StringLibraryTest.csproj
```

The terminal output shows that one test fails, and it provides an error message for the failed test: "Assert.IsFalse failed. Expected for 'Error': false; actual: True". Because of the failure, no strings in the array after "Error" were tested.

Output

```
Starting test execution, please wait...
A total of 1 test files matched the specified pattern.
Failed TestDoesNotStartWithUpper [28 ms]
Error Message:
    Assert.IsFalse failed. Expected for 'Error': false; Actual: True
Stack Trace:
    at StringLibraryTest.UnitTest1.TestDoesNotStartWithUpper() in
C:\ClassLibraryProjects\StringLibraryTest\UnitTest1.cs:line 33

Failed! - Failed:      1, Passed:      2, Skipped:      0, Total:      3,
Duration: 31 ms - StringLibraryTest.dll (net5.0)
```

3. Remove the string "Error" that you added in step 1. Rerun the test and the tests pass.

Test the Release version of the library

Now that the tests have all passed when running the Debug build of the library, run the tests an additional time against the Release build of the library. A number of factors, including compiler optimizations, can sometimes produce different behavior between Debug and Release builds.

1. Run the tests with the Release build configuration:

.NET CLI

```
dotnet test StringLibraryTest/StringLibraryTest.csproj --configuration
Release
```

The tests pass.

Debug tests

If you're using Visual Studio Code as your IDE, you can use the same process shown in [Debug a .NET console application using Visual Studio Code](#) to debug code using your

unit test project. Instead of starting the *ShowCase* app project, open *StringLibraryTest/UnitTest1.cs*, and select **Debug All Tests** between lines 7 and 8. If you're unable to find it, press **Ctrl + Shift + P** to open the command palette and enter **Reload Window**.

Visual Studio Code starts the test project with the debugger attached. Execution will stop at any breakpoint you've added to the test project or the underlying library code.

Additional resources

- [Unit testing in .NET](#)

Next steps

In this tutorial, you unit tested a class library. You can make the library available to others by publishing it to [NuGet](#) as a package. To learn how, follow a NuGet tutorial:

[Create and publish a package using the dotnet CLI](#)

If you publish a library as a NuGet package, others can install and use it. To learn how, follow a NuGet tutorial:

[Install and use a package using the dotnet CLI](#)

A library doesn't have to be distributed as a package. It can be bundled with a console app that uses it. To learn how to publish a console app, see the earlier tutorial in this series:

[Publish a .NET console application using Visual Studio Code](#)

The Visual Studio Code extension C# Dev Kit provides more tools for developing C# apps and libraries:

[C# Dev Kit](#)

 Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

issues and pull requests. For more information, see [our contributor guide](#).

 [Open a documentation issue](#)

 [Provide product feedback](#)

Quickstart: Install and use a package with the dotnet CLI

Article • 08/21/2023

NuGet packages contain compiled binary code that developers make available for other developers to use in their projects. For more information, see [What is NuGet](#). This quickstart describes how to install the popular [Newtonsoft.Json](#) NuGet package into a .NET project by using the [dotnet add package](#) command.

You refer to installed packages in code with a `using <namespace>` directive, where `<namespace>` is often the package name. You can then use the package's API in your project.

Tip

Browse [nuget.org/packages](#) to find packages you can reuse in your own applications. You can search directly at <https://nuget.org>, or find and install packages from within Visual Studio. For more information, see [Find and evaluate NuGet packages for your project](#).

Prerequisites

- The [.NET SDK](#), which provides the `dotnet` command-line tool. Starting in Visual Studio 2017, the dotnet CLI automatically installs with any .NET or .NET Core related workloads.

Create a project

You can install NuGet packages into a .NET project. For this walkthrough, create a simple .NET console project by using the dotnet CLI, as follows:

1. Create a folder named *Nuget.Quickstart* for the project.
2. Open a command prompt and switch to the new folder.
3. Create the project by using the following command:

.NET CLI

```
dotnet new console
```

4. Use `dotnet run` to test the app. You should see the output `Hello, World!`.

Add the Newtonsoft.Json NuGet package

1. Use the following command to install the `Newtonsoft.json` package:

.NET CLI

```
dotnet add package Newtonsoft.Json
```

2. After the command completes, open the `Nuget.Quickstart.csproj` file in Visual Studio to see the added NuGet package reference:

XML

```
<ItemGroup>
  <PackageReference Include="Newtonsoft.Json" Version="13.0.1" />
</ItemGroup>
```

Use the Newtonsoft.Json API in the app

1. In Visual Studio, open the `Program.cs` file and add the following line at the top of the file:

C#

```
using Newtonsoft.Json;
```

2. Add the following code to replace the `Console.WriteLine("Hello, World!");` statement:

C#

```
namespace Nuget.Quickstart
{
    public class Account
    {
        public string Name { get; set; }
        public string Email { get; set; }
        public DateTime DOB { get; set; }
    }
}
```

```
internal class Program
{
    static void Main(string[] args)
    {
        Account account = new Account
        {
            Name = "John Doe",
            Email = "john@nuget.org",
            DOB = new DateTime(1980, 2, 20, 0, 0, 0,
DateTimeKind.Utc),
        };

        string json = JsonConvert.SerializeObject(account,
Formatting.Indented);
        Console.WriteLine(json);
    }
}
```

3. Save the file, then build and run the app by using the `dotnet run` command. The output is the JSON representation of the `Account` object in the code:

Output

```
{
  "Name": "John Doe",
  "Email": "john@nuget.org",
  "DOB": "1980-02-20T00:00:00Z"
}
```

Congratulations on installing and using your first NuGet package!

Related video

<https://learn.microsoft.com/shows/NuGet-101/Install-and-Use-a-NuGet-Package-with-the-NET-CLI-3-of-5/player>

Find more NuGet videos on [Channel 9](#) and [YouTube](#).

Next steps

Learn more about installing and using NuGet packages with the dotnet CLI:

[Install and use packages by using the dotnet CLI](#)

- Overview and workflow of package consumption

- Find and choose packages
- Package references in project files

Quickstart: Create and publish a package with the dotnet CLI

Article • 08/21/2023

This quickstart shows you how to quickly create a NuGet package from a .NET class library and publish it to nuget.org by using the .NET command-line interface, or [dotnet CLI](#).

Prerequisites

- The [.NET SDK](#), which provides the dotnet command-line tool. Starting in Visual Studio 2017, the dotnet CLI automatically installs with any .NET or .NET Core related workloads.
- A free account on nuget.org. Follow the instructions at [Add a new individual account](#).

Create a class library project

You can use an existing .NET Class Library project for the code you want to package, or create a simple project as follows:

1. Create a folder named *AppLogger*.
2. Open a command prompt and switch to the *AppLogger* folder. All the dotnet CLI commands in this quickstart run on the current folder by default.
3. Enter `dotnet new classlib`, which creates a project with the current folder name.

For more information, see [dotnet new](#).

Add package metadata to the project file

Every NuGet package has a manifest that describes the package's contents and dependencies. In the final package, the manifest is a *.nuspec* file, which uses the NuGet metadata properties you include in the project file.

Open the *.csproj*, *.fproj*, or *.vbproj* project file, and add the following properties inside the existing `<PropertyGroup>` tag. Use your own values for name and company, and replace the package identifier with a unique value.

XML

```
<PackageId>Contoso.08.28.22.001.Test</PackageId>
<Version>1.0.0</Version>
<Authors>your_name</Authors>
<Company>your_company</Company>
```

ⓘ Important

The package identifier must be unique across nuget.org and other package sources. Publishing makes the package publicly visible, so if you use the example AppLogger library or other test library, use a unique name that includes `Sample` or `Test`.

You can add any optional properties described in [NuGet metadata properties](#).

ⓘ Note

For packages you build for public consumption, pay special attention to the `PackageTags` property. Tags help others find your package and understand what it does.

Run the pack command

To build a NuGet package or `.nupkg` file from the project, run the [dotnet pack](#) command, which also builds the project automatically.

.NET CLI

```
dotnet pack
```

The output shows the path to the `.nupkg` file:

Output

```
MSBuild version 17.3.0+92e077650 for .NET
Determining projects to restore...
Restored C:\Users\myname\source\repos\AppLogger\AppLogger.csproj (in 64
ms).
AppLogger ->
C:\Users\myname\source\repos\AppLogger\bin\Debug\net6.0\AppLogger.dll
Successfully created package
```

```
'C:\Users\myname\source\repos\AppLogger\bin\Debug\Contoso.08.28.22.001.Test.  
1.0.0.nupkg'.
```

Automatically generate package on build

To automatically run `dotnet pack` whenever you run `dotnet build`, add the following line to your project file within `<PropertyGroup>`:

XML

```
<GeneratePackageOnBuild>true</GeneratePackageOnBuild>
```

Publish the package

Publish your `.nupkg` file to nuget.org by using the `dotnet nuget push` command with an API key you get from nuget.org.

① Note

- Nuget.org scans all uploaded packages for viruses and rejects the packages if it finds any viruses. Nuget.org also scans all existing listed packages periodically.
- Packages you publish to nuget.org are publicly visible to other developers unless you unlist them. To host packages privately, see [Host your own NuGet feeds](#).

Get your API key

1. [Sign into your nuget.org account](#) or [create an account](#) if you don't have one already.
2. Select your user name at upper right, and then select **API Keys**.
3. Select **Create**, and provide a name for your key.
4. Under **Select Scopes**, select **Push**.
5. Under **Select Packages > Glob Pattern**, enter `*`.
6. Select **Create**.

7. Select **Copy** to copy the new key.

The screenshot shows a UI for managing API keys. At the top, a yellow banner displays a warning: "⚠ Your API key has been regenerated. Make sure to copy your new API key now using the **Copy** button below. You will not be able to do so again." Below the banner, the key details are shown: "test-key1", "Expires in a year", "Push new packages and package versions", "Package owner: NuGet-test", "Glob pattern: *". A red box highlights the "Copy" button, which is located next to "Edit", "Regenerate", and "Delete" buttons.

ⓘ Important

- Always keep your API key a secret. The API key is like a password that allows anyone to manage packages on your behalf. Delete or regenerate your API key if it's accidentally revealed.
- Save your key in a secure location, because you can't copy the key again later. If you return to the API key page, you need to regenerate the key to copy it. You can also remove the API key if you no longer want to push packages.

Scoping lets you create separate API keys for different purposes. Each key has an expiration timeframe, and you can scope the key to specific packages or glob patterns. You also scope each key to specific operations: Push new packages and package versions, push only new package versions, or unlist.

Through scoping, you can create API keys for different people who manage packages for your organization so they have only the permissions they need.

For more information, see [scoped API keys](#).

Publish with `dotnet nuget push`

From the folder that contains the `.nupkg` file, run the following command. Specify your `.nupkg` filename, and replace the key value with your API key.

.NET CLI

```
dotnet nuget push Contoso.08.28.22.001.Test.1.0.0.nupkg --api-key  
qz2jga8p13dvn2akksyquwcs9ygggg4exypy3bhxy6w6x6 --source  
https://api.nuget.org/v3/index.json
```

The output shows the results of the publishing process:

Output

```
Pushing Contoso.08.28.22.001.Test.1.0.0.nupkg to  
'https://www.nuget.org/api/v2/package'...  
PUT https://www.nuget.org/api/v2/package/  
warn : All published packages should have license information specified.  
Learn more: https://aka.ms/nuget/authoring-best-practices#licensing.  
Created https://www.nuget.org/api/v2/package/ 1221ms  
Your package was pushed.
```

For more information, see [dotnet nuget push](#).

ⓘ Note

If you want to avoid your test package being live on nuget.org, you can push to the nuget.org test site at <https://int.nugettest.org>. Note that packages uploaded to int.nugettest.org might not be preserved.

Publish errors

Errors from the `push` command typically indicate the problem. For example, you might have forgotten to update the version number in your project, so you're trying to publish a package that already exists.

You also see errors if your API key is invalid or expired, or if you try to publish a package using an identifier that already exists on the host. Suppose, for example, the identifier `AppLogger-test` already exists on nuget.org. If you try to publish a package with that identifier, the `push` command gives the following error:

Output

```
Response status code does not indicate success: 403 (The specified API key  
is invalid,  
has expired, or does not have permission to access the specified package.).
```

If you get this error, check that you're using a valid API key that hasn't expired. If you are, the error indicates the package identifier already exists on the host. To fix the error,

change the package identifier to be unique, rebuild the project, recreate the `.nupkg` file, and retry the `push` command.

Manage the published package

When your package successfully publishes, you receive a confirmation email. To see the package you just published, on nuget.org, select your user name at upper right, and then select **Manage Packages**.

Note

It might take awhile for your package to be indexed and appear in search results where others can find it. During that time, your package appears under **Unlisted Packages**, and the package page shows the following message:

 **This package has not been published yet.** It will appear in search results and will be available for install/restore after both validation and indexing are complete. Package validation and indexing may take up to an hour. [Read more](#).

You've now published a NuGet package to nuget.org that other developers can use in their projects.

If you've created a package that isn't useful (such as this sample package that was created with an empty class library), or you decide you don't want the package to be visible, you can *unlist* the package to hide it from search results:

1. After the package appears under **Published Packages** on the **Manage Packages** page, select the pencil icon next to the package listing.

| Published Packages | | | | | 1 package / 0 downloads |
|---|--------|---------------------------|-----------|----------------|---|
| Package ID | Owners | Signing Owner | Downloads | Latest Version | |
|  Contoso.08.28.22.001.Test | Test | username (0 certificates) | 0 | 1.0.0 |  |

2. On the next page, select **Listing**, deselect the **List in search results** checkbox, and then select **Save**.

✓ Listing

Select version

1.0.0 (Latest)

List or unlist version

ⓘ You can control how your packages are listed using the checkbox below. As per [policy](#), permanent deletion is not supported as it would break every project depending on the availability of the package. For more assistance, [Contact Support](#).

List in search results.

Unlisted packages cannot be installed directly and do not show up in search results.

Save

The package now appears under **Unlisted Packages** in **Manage Packages** and no longer appears in search results.

ⓘ Note

To avoid your test package being live on nuget.org, you can push to the nuget.org test site at <https://int.nugettest.org>. Note that packages uploaded to int.nugettest.org might not be preserved.

Congratulations on creating and publishing your first NuGet package!

Related video

<https://learn.microsoft.com/shows/NuGet-101/Create-and-Publish-a-NuGet-Package-with-the-NET-CLI-5-of-5/player>

Find more NuGet videos on [Channel 9](#) and [YouTube](#).

Next steps

See more details about how to create packages with the dotnet CLI:

Create a NuGet package with the dotnet CLI

Get more information about creating and publishing NuGet packages:

- [Publish a package](#)
- [Prerelease packages](#)
- [Support multiple target frameworks](#)
- [Package versioning](#)
- [Add a license expression or file](#)
- [Create localized packages](#)
- [Create symbol packages](#)
- [Sign packages](#)

Tutorial: Create a .NET console application using Visual Studio for Mac

Article • 09/08/2023

This tutorial shows how to create and run a .NET console application using Visual Studio for Mac.

Important

Microsoft has announced the retirement of Visual Studio for Mac. Visual Studio for Mac will no longer be supported starting August 31, 2024. Alternatives include:

- Visual Studio Code with the [C# Dev Kit](#) and related extensions, such as [.NET MAUI](#) and [Unity](#).
- Visual Studio running on Windows in a VM on Mac.
- Visual Studio running on Windows in a [VM in the Cloud](#).

For more information, see [Visual Studio for Mac retirement announcement](#).

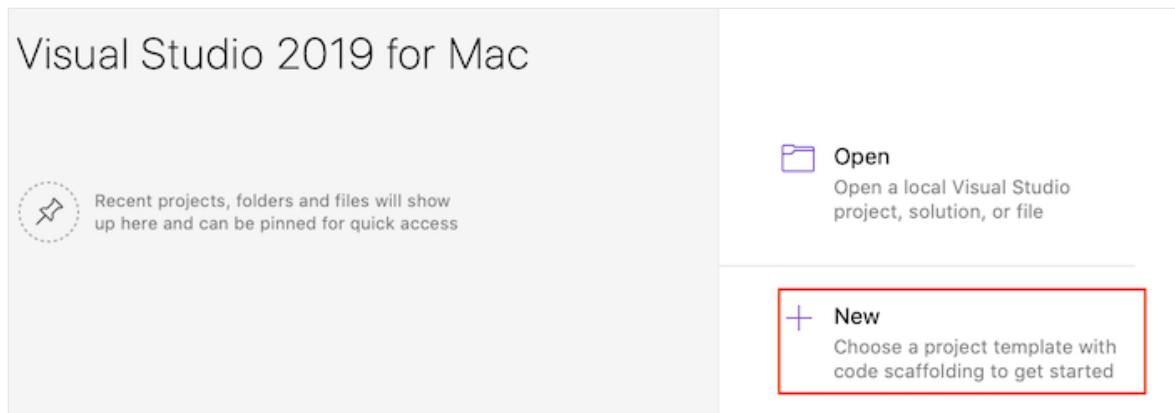
Prerequisites

- [Visual Studio for Mac version 8.8 or later](#). Select the option to install .NET Core. Installing Xamarin is optional for .NET development. For more information, see the following resources:
 - [Tutorial: Install Visual Studio for Mac](#).
 - [Supported macOS versions](#).
 - [.NET versions supported by Visual Studio for Mac](#).

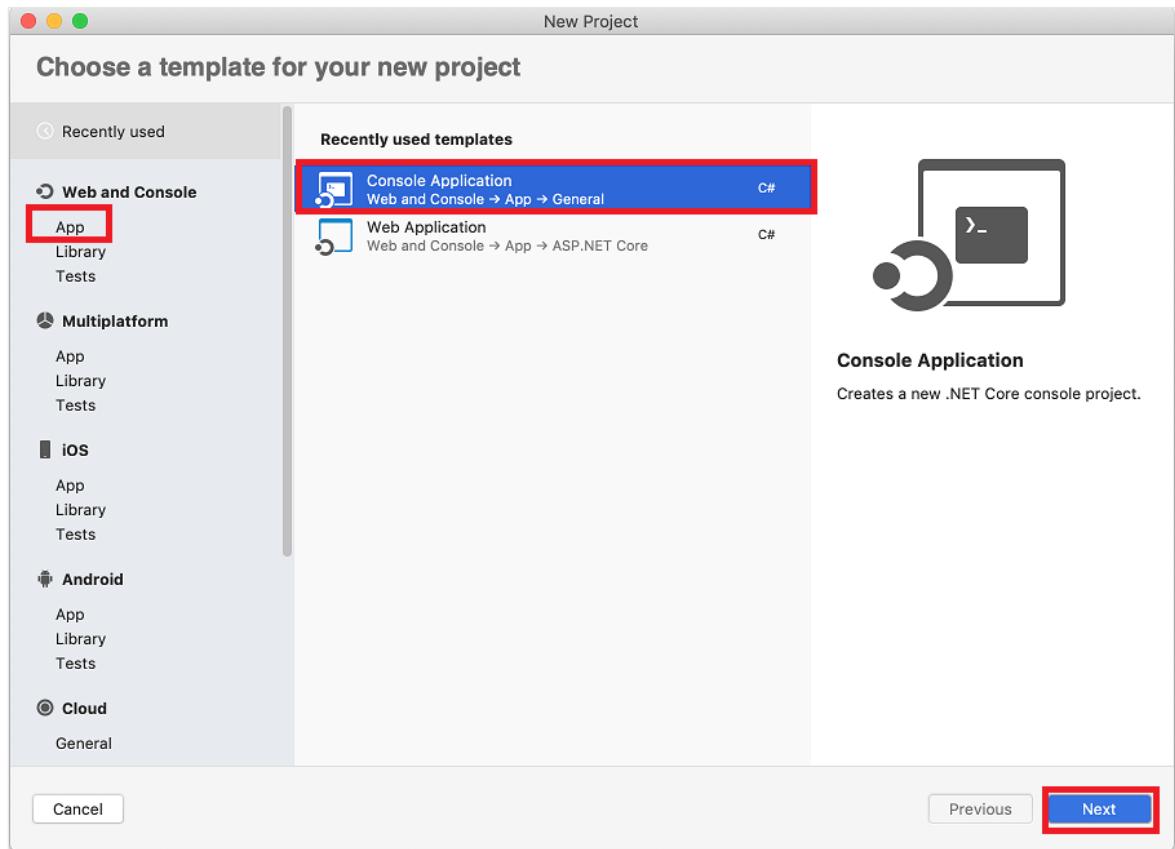
Create the app

1. Start Visual Studio for Mac.
2. Select **New** in the start window.

Visual Studio 2019 for Mac

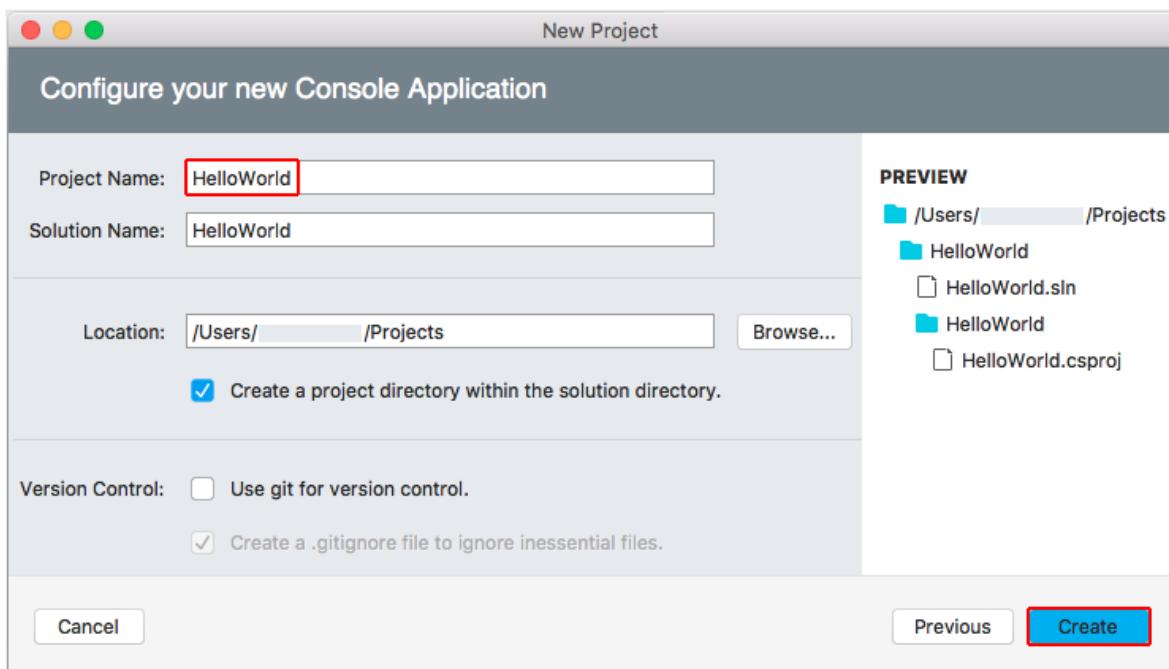


3. In the **New Project** dialog, select **App** under the **Web and Console** node. Select the **Console Application** template, and select **Next**.



4. In the **Target Framework** drop-down of the **Configure your new Console Application** dialog, select **.NET 5.0**, and select **Next**.

5. Type "HelloWorld" for the **Project Name**, and select **Create**.



The template creates a simple "Hello World" application. It calls the [Console.WriteLine\(String\)](#) method to display "Hello World!" in the terminal window.

The template code defines a class, `Program`, with a single method, `Main`, that takes a `String` array as an argument:

```
C#  
  
using System;  
  
namespace HelloWorld  
{  
    class Program  
    {  
        static void Main(string[] args)  
        {  
            Console.WriteLine("Hello World!");  
        }  
    }  
}
```

`Main` is the application entry point, the method that's called automatically by the runtime when it launches the application. Any command-line arguments supplied when the application is launched are available in the `args` array.

Run the app

1. Press (`option` + `command` + `enter`) to run the app without debugging.

The screenshot shows the Visual Studio for Mac interface. At the top, the menu bar includes options like File, Edit, View, Search, Project, Build, and Run. Below the menu is a toolbar with icons for back, forward, and run/debug. A status bar at the bottom indicates "Debug > Default". A message box in the center says, "We'd love to hear about your .NET Core experience in Visual Studio for Mac." The main workspace contains a code editor titled "Program.cs" with the following C# code:

```
1  using System;
2
3  namespace HelloWorld
4  {
5      class Program
6      {
7          static void Main(string[] args)
8          {
9              Console.WriteLine("Hello World!");
10         }
11     }
12 }
13
```

Below the code editor is a terminal window titled "Terminal - HelloWorld" which displays the output "Hello World!". The output text is highlighted with a red border.

2. Close the Terminal window.

Enhance the app

Enhance the application to prompt the user for their name and display it along with the date and time.

1. In *Program.cs*, replace the contents of the `Main` method, which is the line that calls `Console.WriteLine`, with the following code:

C#

```
Console.WriteLine("What is your name?");
var name = Console.ReadLine();
var currentDate = DateTime.Now;
Console.WriteLine($"{Environment.NewLine}Hello, {name}, on
{currentDate:d} at {currentDate:t}!");
Console.WriteLine($"{Environment.NewLine}Press any key to exit...");
Console.ReadKey(true);
```

This code displays a prompt in the console window and waits until the user enters a string followed by the `enter` key. It stores this string in a variable named `name`. It also retrieves the value of the `DateTime.Now` property, which contains the current local time, and assigns it to a variable named `currentDate`. And it displays these values in the console window. Finally, it displays a prompt in the console window and calls the `Console.ReadKey(Boolean)` method to wait for user input.

`NewLine` is a platform-independent and language-independent way to represent a line break. Alternatives are `\n` in C# and `vbcrlf` in Visual Basic.

The dollar sign (\$) in front of a string lets you put expressions such as variable names in curly braces in the string. The expression value is inserted into the string in place of the expression. This syntax is referred to as [interpolated strings](#).

2. Press `⌘ ⌘ ⌘` (`option + command + enter`) to run the app.
3. Respond to the prompt by entering a name and pressing `enter`.

```
Terminal - HelloWorld
What is your name?
Nancy
Hello, Nancy, on 6/4/2020 at 3:42 PM!
Press any key to exit...
```

4. Close the terminal.

Next steps

In this tutorial, you created a .NET console application. In the next tutorial, you debug the app.

[Debug a .NET console application using Visual Studio for Mac](#)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

Tutorial: Debug a .NET console application using Visual Studio for Mac

Article • 09/08/2023

ⓘ Important

Microsoft has announced the retirement of Visual Studio for Mac. Visual Studio for Mac will no longer be supported starting August 31, 2024. Alternatives include:

- Visual Studio Code with the [C# Dev Kit](#) and related extensions, such as [.NET MAUI](#) and [Unity](#).
- Visual Studio running on Windows in a VM on Mac.
- Visual Studio running on Windows in a [VM in the Cloud](#).

For more information, see [Visual Studio for Mac retirement announcement](#).

This tutorial introduces the debugging tools available in Visual Studio for Mac.

Prerequisites

- This tutorial works with the console app that you create in [Create a .NET console application using Visual Studio for Mac](#).

Use Debug build configuration

Debug and *Release* are Visual Studio's built-in build configurations. You use the *Debug* build configuration for debugging and the *Release* configuration for the final release distribution.

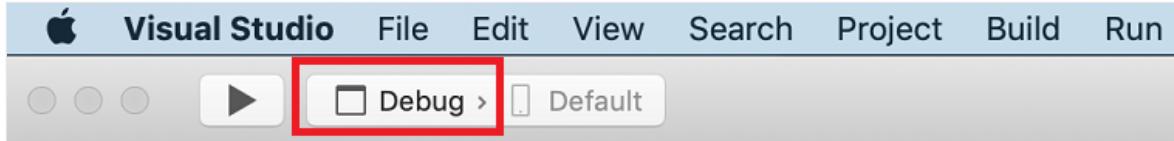
In the *Debug* configuration, a program compiles with full symbolic debug information and no optimization. Optimization complicates debugging, because the relationship between source code and generated instructions is more complex. The *release* configuration of a program has no symbolic debug information and is fully optimized.

By default, Visual Studio for Mac uses the *Debug* build configuration, so you don't need to change it before debugging.

1. Start Visual Studio for Mac.

2. Open the project that you created in [Create a .NET console application using Visual Studio for Mac](#).

The current build configuration is shown on the toolbar. The following toolbar image shows that Visual Studio is configured to compile the Debug version of the app:

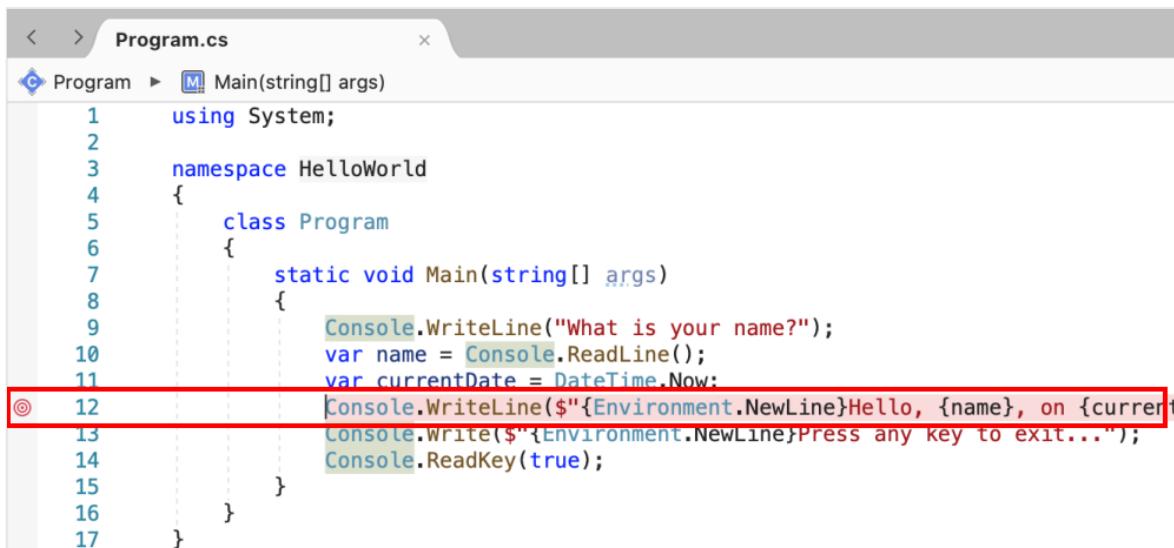


Set a breakpoint

A *breakpoint* temporarily interrupts the execution of the application before the line with the breakpoint is executed.

1. Set a breakpoint on the line that displays the name, date, and time. To do that, place the cursor in the line of code and press $\text{⌘} \backslash$ ($\text{command} + \backslash$). Another way to set a breakpoint is by selecting **Debug > Toggle Breakpoint** from the menu.

Visual Studio indicates the line on which the breakpoint is set by highlighting it and displaying a red dot in the left margin.



2. Press $\text{⌘} \leftarrow$ ($\text{command} + \text{enter}$) to start the program in debugging mode. Another way to start debugging is by choosing **Debug > Start Debugging** from the menu.
3. Enter a string in the terminal window when the program prompts for a name, and then press **enter**.
4. Program execution stops when it reaches the breakpoint, before the `Console.WriteLine` method executes.

```
1  using System;
2
3  namespace HelloWorld
4  {
5      class Program
6      {
7          static void Main(string[] args)
8          {
9              Console.WriteLine("What is your name?");
10             var name = Console.ReadLine();
11             var currentDate = DateTime.Now;
12             Console.WriteLine($"{Environment.NewLine}Hello, {name}, on {currentDate}");
13             Console.WriteLine($"{Environment.NewLine}Press any key to exit...");
14             Console.ReadKey(true);
15         }
16     }
17 }
```

Use the Immediate window

The **Immediate** window lets you interact with the application you're debugging. You can interactively change the value of variables to see how it affects your program.

1. If the **Immediate** window is not visible, display it by choosing **View > Debug Windows > Immediate**.
2. Enter `name = "Gracie"` in the **Immediate** window and press `enter`.
3. Enter `currentDate = currentDate.AddDays(1)` in the **Immediate** window and press `enter`.

The **Immediate** window displays the new value of the string variable and the properties of the `DateTime` value.

```
[-] Immediate
name = "Gracie"
"Gracie"
currentDate = currentDate.AddDays(1)
{5/27/2021 11:15:29 AM}
    Date: {5/27/2021 12:00:00 AM}
    Day: 27
    DayOfWeek: System.DayOfWeek.Thursday
    DayOfYear: 147
    Hour: 11
    Kind: System.DateTimeKind.Local
    Millisecond: 799
    Minute: 15
    Month: 5
    Second: 29
    Ticks: 637577109297996840
    TimeOfDay: {11:15:29.7996840}
    Year: 2021
Static members:
Non-Public members:
```

The **Locals** window displays the values of variables that are defined in the currently executing method. The values of the variables that you just changed are updated in the **Locals** window.

| Breakpoints | Locals | Watch | Threads |
|-------------|-------------|-------------------------|-----------------|
| Name | Value | Type | |
| | args | {string[0]} | string[] |
| | name | Gracie | string |
| ▶ | currentDate | {5/27/2021 11:15:29 AM} | System.DateTime |

4. Press **⌘ ↵** (**command** + **enter**) to continue debugging.

The values displayed in the terminal correspond to the changes you made in the **Immediate** window.

If you don't see the Terminal, select **Terminal - HelloWorld** in the bottom navigation bar.



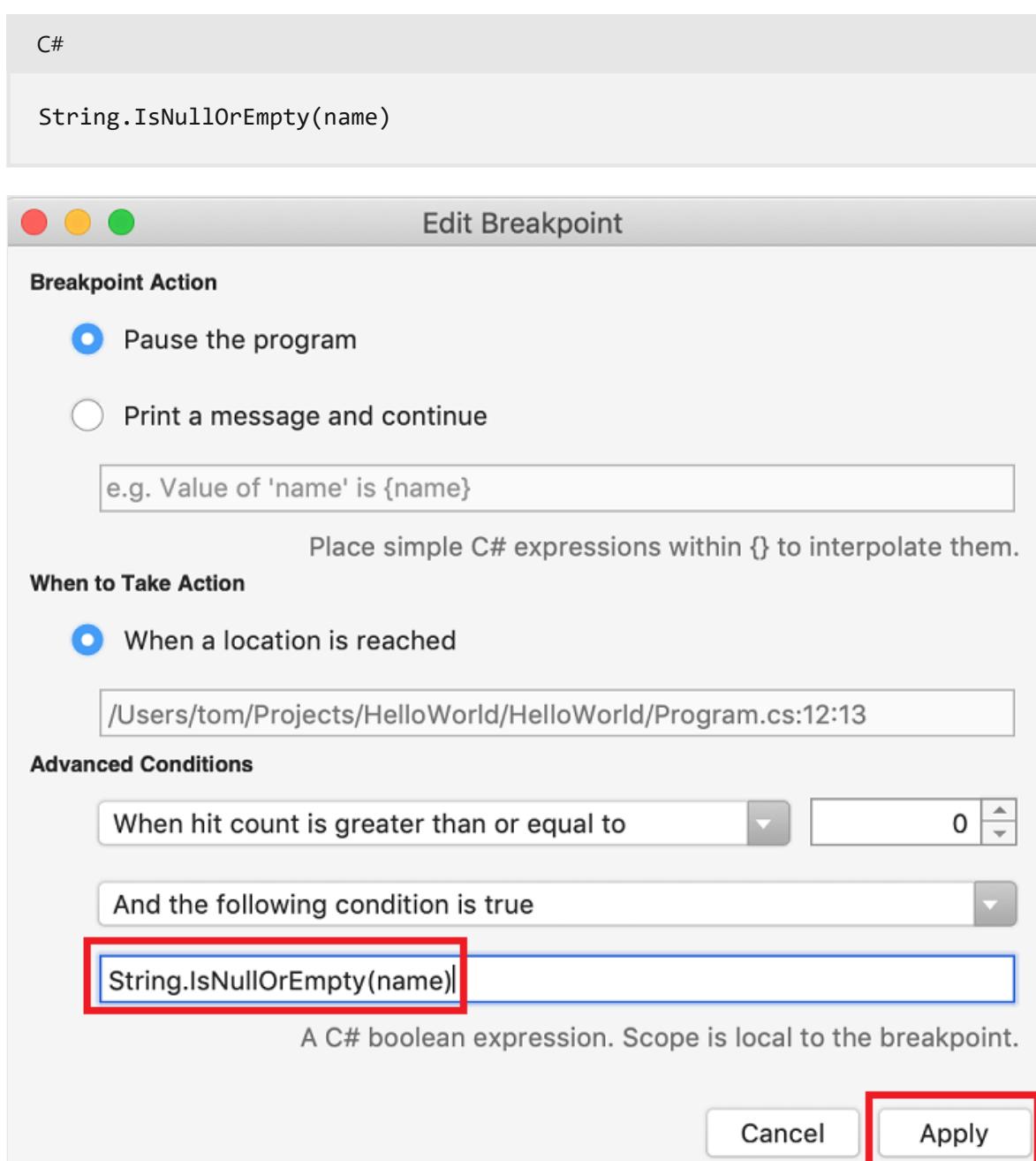
5. Press any key to exit the program.

6. Close the terminal window.

Set a conditional breakpoint

The program displays a string that the user enters. What happens if the user doesn't enter anything? You can test this with a useful debugging feature called a *conditional breakpoint*.

1. `ctrl`-click on the red dot that represents the breakpoint. In the context menu, select **Edit Breakpoint**.
2. In the **Edit Breakpoint** dialog, enter the following code in the field that follows **And the following condition is true**, and select **Apply**.



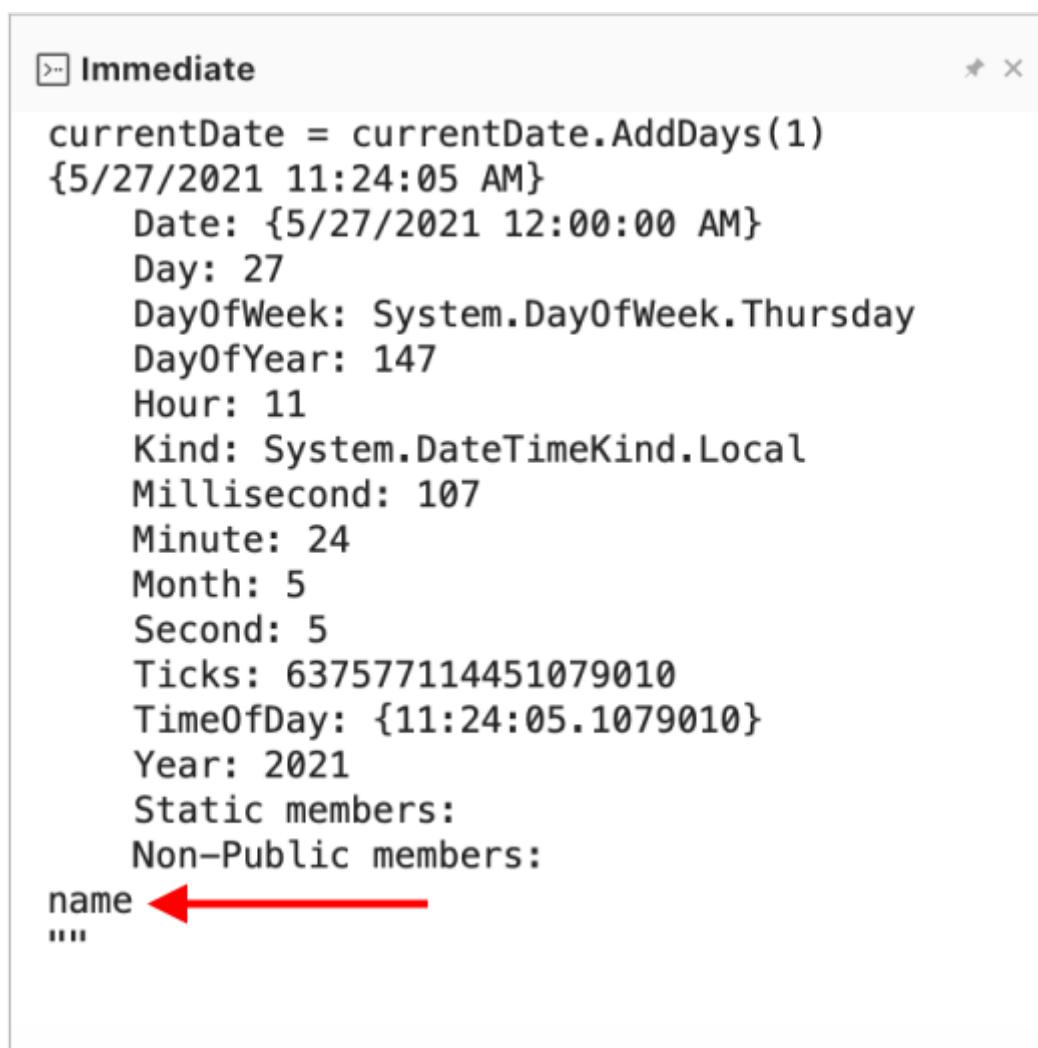
Each time the breakpoint is hit, the debugger calls the `String.IsNullOrEmpty(name)` method, and it breaks on this line only if the method call returns `true`.

Instead of a conditional expression, you can specify a *hit count*, which interrupts program execution before a statement is executed a specified number of times.

3. Press `⌘ ↵` (`command + enter`) to start debugging.
4. In the terminal window, press `enter` when prompted to enter your name.

Because the condition you specified (`name` is either `null` or `String.Empty`) has been satisfied, program execution stops when it reaches the breakpoint.

5. Select the **Locals** window, which shows the values of variables that are local to the currently executing method. In this case, `Main` is the currently executing method. Observe that the value of the `name` variable is `""`, that is, `String.Empty`.
6. You can also see that the value is an empty string by entering the `name` variable name in the **Immediate** window and pressing `enter`.



The screenshot shows the Immediate window with the following output:

```
currentDate = currentDate.AddDays(1)
{5/27/2021 11:24:05 AM}
  Date: {5/27/2021 12:00:00 AM}
  Day: 27
  DayOfWeek: System.DayOfWeek.Thursday
  DayOfYear: 147
  Hour: 11
  Kind: System.DateTimeKind.Local
  Millisecond: 107
  Minute: 24
  Month: 5
  Second: 5
  Ticks: 637577114451079010
  TimeOfDay: {11:24:05.1079010}
  Year: 2021
  Static members:
  Non-Public members:
name ←
"
```

A red arrow points to the word "name" in the Immediate window output.

7. Press `⌘ ↵` (`command + enter`) to continue debugging.

8. In the terminal window, press any key to exit the program.

9. Close the terminal window.

10. Clear the breakpoint by clicking on the red dot in the left margin of the code window. Another way to clear a breakpoint is by choosing **Debug > Toggle Breakpoint** while the line of code is selected.

Step through a program

Visual Studio also allows you to step line by line through a program and monitor its execution. Ordinarily, you'd set a breakpoint and follow program flow through a small part of your program code. Since this program is small, you can step through the entire program.

1. Set a breakpoint on the curly brace that marks the start of the `Main` method (press `command + \`).

2. Press `⌘ ↵` (`command + enter`) to start debugging.

Visual Studio stops on the line with the breakpoint.

3. Press `↑ ⌘ I` (`shift + command + I`) or select **Debug > Step Into** to advance one line.

Visual Studio highlights and displays an arrow beside the next line of execution.

The screenshot shows the `Program.cs` file in the Visual Studio code editor. A red circular breakpoint icon is positioned to the left of the first brace on line 8. The cursor is located on line 9, just before the opening brace of the `Main` method. The code is as follows:

```
1  using System;
2
3  namespace HelloWorld
4  {
5      class Program
6      {
7          static void Main(string[] args)
8          {
9              Console.WriteLine("What is your name?");
10             var name = Console.ReadLine();
11             var currentDate = DateTime.Now;
12             Console.WriteLine($"{Environment.NewLine}Hello, {name}, on {currentDate}");
13             Console.WriteLine($"{Environment.NewLine}Press any key to exit...");
14             Console.ReadKey(true);
15         }
16     }
17 }
```

At this point, the **Locals** window shows that the `args` array is empty, and `name` and `currentDate` have default values. In addition, Visual Studio has opened a blank terminal.

4. Press    (`shift`+`command`+`I`).

Visual Studio highlights the statement that includes the `name` variable assignment. The **Locals** window shows that `name` is `null`, and the terminal displays the string "What is your name?".

5. Respond to the prompt by entering a string in the console window and pressing `enter`.

6. Press    (`shift`+`command`+`I`).

Visual Studio highlights the statement that includes the `currentDate` variable assignment. The **Locals** window shows the value returned by the call to the [Console.ReadLine](#) method. The terminal displays the string you entered at the prompt.

7. Press    (`shift`+`command`+`I`).

The **Locals** window shows the value of the `currentDate` variable after the assignment from the [DateTime.Now](#) property. The terminal is unchanged.

8. Press    (`shift`+`command`+`I`).

Visual Studio calls the [Console.WriteLine\(String, Object, Object\)](#) method. The terminal displays the formatted string.

9. Press    (`shift`+`command`+`U`) or select **Run > Step Out**.

The terminal displays a message and waits for you to press a key.

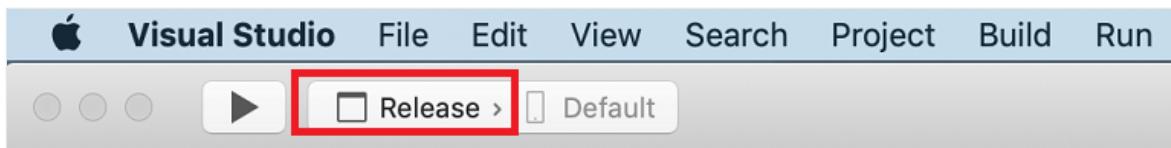
10. Press any key to exit the program.

Use Release build configuration

Once you've tested the Debug version of your application, you should also compile and test the Release version. The Release version incorporates compiler optimizations that can negatively affect the behavior of an application. For example, compiler optimizations that are designed to improve performance can create race conditions in multithreaded applications.

To build and test the Release version of the console application, do the following steps:

1. Change the build configuration on the toolbar from **Debug** to **Release**.



2. Press `⌥ ⌘ ⏎` (`option + command + enter`) to run without debugging.

Next steps

In this tutorial, you used Visual Studio debugging tools. In the next tutorial, you publish a deployable version of the app.

[Publish a .NET console application using Visual Studio for Mac](#)

 Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

Tutorial: Publish a .NET console application using Visual Studio for Mac

Article • 09/08/2023

ⓘ Important

Microsoft has announced the retirement of Visual Studio for Mac. Visual Studio for Mac will no longer be supported starting August 31, 2024. Alternatives include:

- Visual Studio Code with the [C# Dev Kit](#) and related extensions, such as [.NET MAUI](#) and [Unity](#).
- Visual Studio running on Windows in a VM on Mac.
- Visual Studio running on Windows in a [VM in the Cloud](#).

For more information, see [Visual Studio for Mac retirement announcement](#).

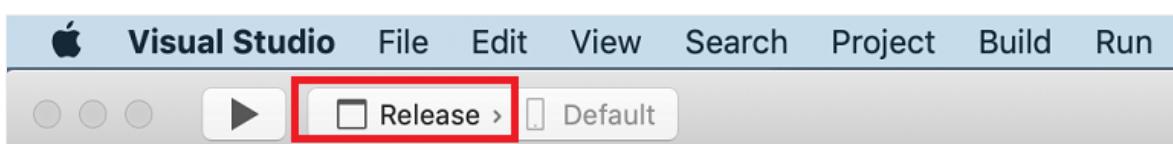
This tutorial shows how to publish a console app so that other users can run it. Publishing creates the set of files that are needed to run your application. To deploy the files, copy them to the target machine.

Prerequisites

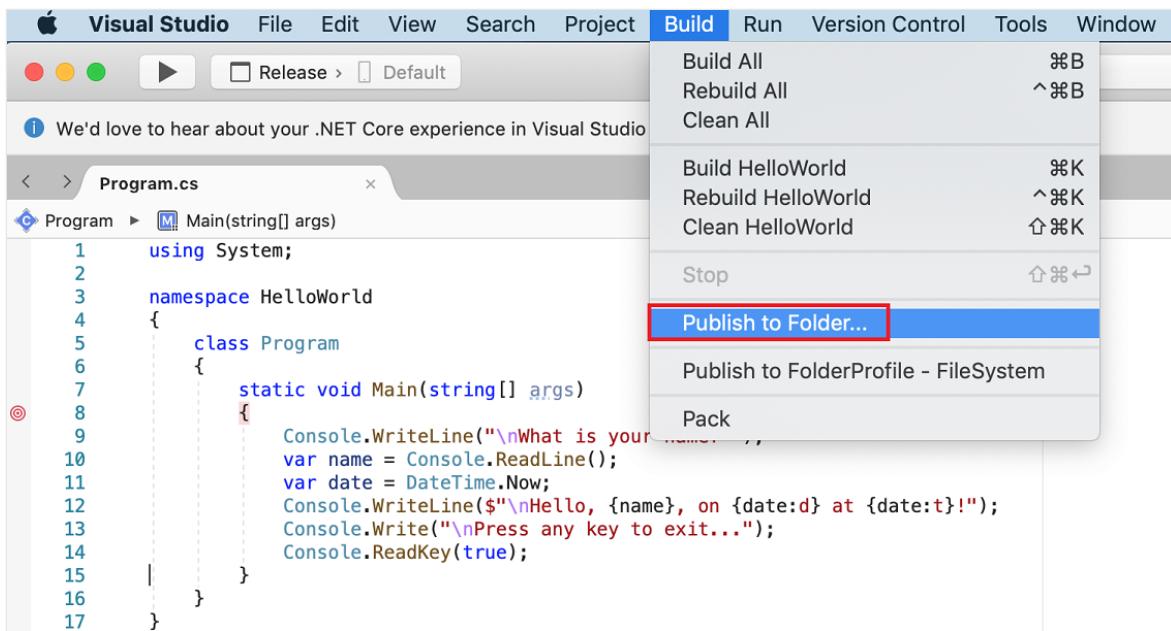
- This tutorial works with the console app that you create in [Create a .NET console application using Visual Studio for Mac](#).

Publish the app

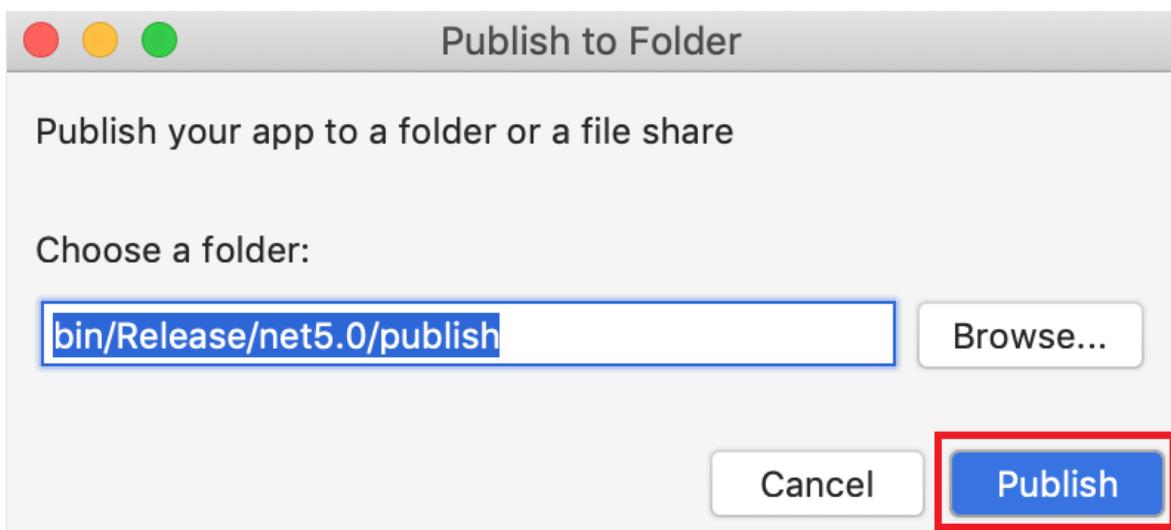
1. Start Visual Studio for Mac.
2. Open the HelloWorld project that you created in [Create a .NET console application using Visual Studio for Mac](#).
3. Make sure that Visual Studio is building the Release version of your application. If necessary, change the build configuration setting on the toolbar from **Debug** to **Release**.



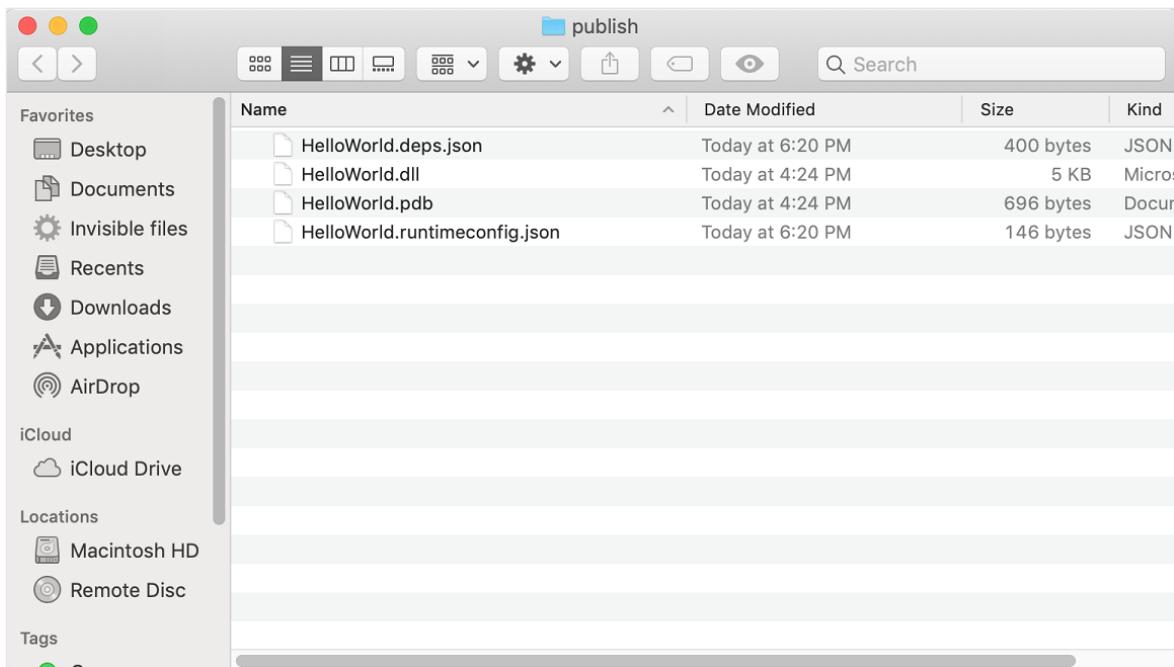
4. From the main menu, choose Build > Publish to Folder....



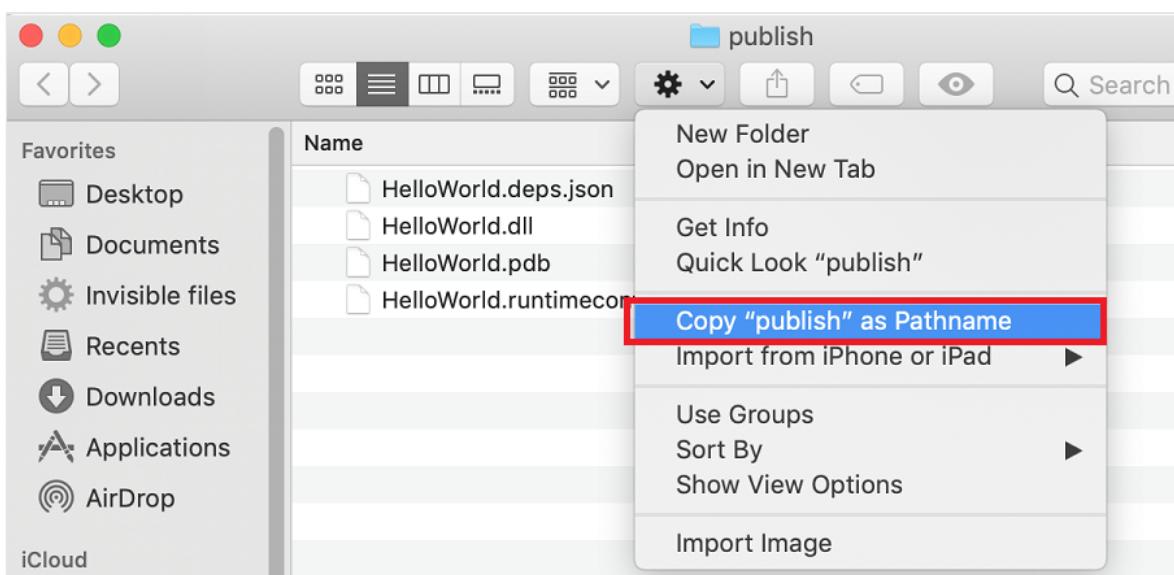
5. In the Publish to Folder dialog, select Publish.



The publish folder opens, showing the files that were created.



6. Select the gear icon, and select **Copy "publish" as Pathname** from the context menu.



Inspect the files

The publishing process creates a framework-dependent deployment, which is a type of deployment where the published application runs on a machine that has the .NET runtime installed. Users can run the published app by running the `dotnet HelloWorld.dll` command from a command prompt.

As the preceding image shows, the published output includes the following files:

- *HelloWorld.deps.json*

This is the application's runtime dependencies file. It defines the .NET components and the libraries (including the dynamic link library that contains your application) needed to run the app. For more information, see [Runtime configuration files](#).

- *HelloWorld.dll*

This is the [framework-dependent deployment](#) version of the application. To execute this dynamic link library, enter `dotnet HelloWorld.dll` at a command prompt. This method of running the app works on any platform that has the .NET runtime installed.

- *HelloWorld.pdb* (optional for deployment)

This is the debug symbols file. You aren't required to deploy this file along with your application, although you should save it in the event that you need to debug the published version of your application.

- *HelloWorld.runtimeconfig.json*

This is the application's runtime configuration file. It identifies the version of .NET that your application was built to run on. You can also add configuration options to it. For more information, see [.NET runtime configuration settings](#).

Run the published app

1. Open a terminal and navigate to the *publish* folder. To do that, enter `cd` and then paste the path that you copied earlier. For example:

```
Console
```

```
cd ~/Projects/HelloWorld/HelloWorld/bin/Release/net5.0/publish/
```

2. Run the app by using the `dotnet` command:

- a. Enter `dotnet HelloWorld.dll` and press `enter`.

- b. Enter a name in response to the prompt, and press any key to exit.

Additional resources

- [.NET application deployment](#)
- [Publish .NET apps with the .NET CLI](#)
- [dotnet publish](#)

- Tutorial: Publish a .NET console application using Visual Studio Code
- Use the .NET SDK in continuous integration (CI) environments

Next steps

In this tutorial, you published a console app. In the next tutorial, you create a class library.

[Create a .NET library using Visual Studio for Mac](#)

 Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

 .NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

Tutorial: Create a .NET class library using Visual Studio for Mac

Article • 09/08/2023

ⓘ Important

Microsoft has announced the retirement of Visual Studio for Mac. Visual Studio for Mac will no longer be supported starting August 31, 2024. Alternatives include:

- Visual Studio Code with the [C# Dev Kit](#) and related extensions, such as [.NET MAUI](#) and [Unity](#).
- Visual Studio running on Windows in a VM on Mac.
- Visual Studio running on Windows in a [VM in the Cloud](#).

For more information, see [Visual Studio for Mac retirement announcement](#).

In this tutorial, you create a class library that contains a single string-handling method.

A *class library* defines types and methods that are called by an application. If the library targets .NET Standard 2.0, it can be called by any .NET implementation (including .NET Framework) that supports .NET Standard 2.0. If the library targets .NET 5, it can be called by any application that targets .NET 5. This tutorial shows how to target .NET 5.

ⓘ Note

Your feedback is highly valued. There are two ways you can provide feedback to the development team on Visual Studio for Mac:

- In Visual Studio for Mac, select **Help > Report a Problem** from the menu or **Report a Problem** from the Welcome screen, which opens a window for filing a bug report. You can track your feedback in the [Developer Community](#) portal.
- To make a suggestion, select **Help > Provide a Suggestion** from the menu or **Provide a Suggestion** from the Welcome screen, which takes you to the [Visual Studio for Mac Developer Community webpage](#).

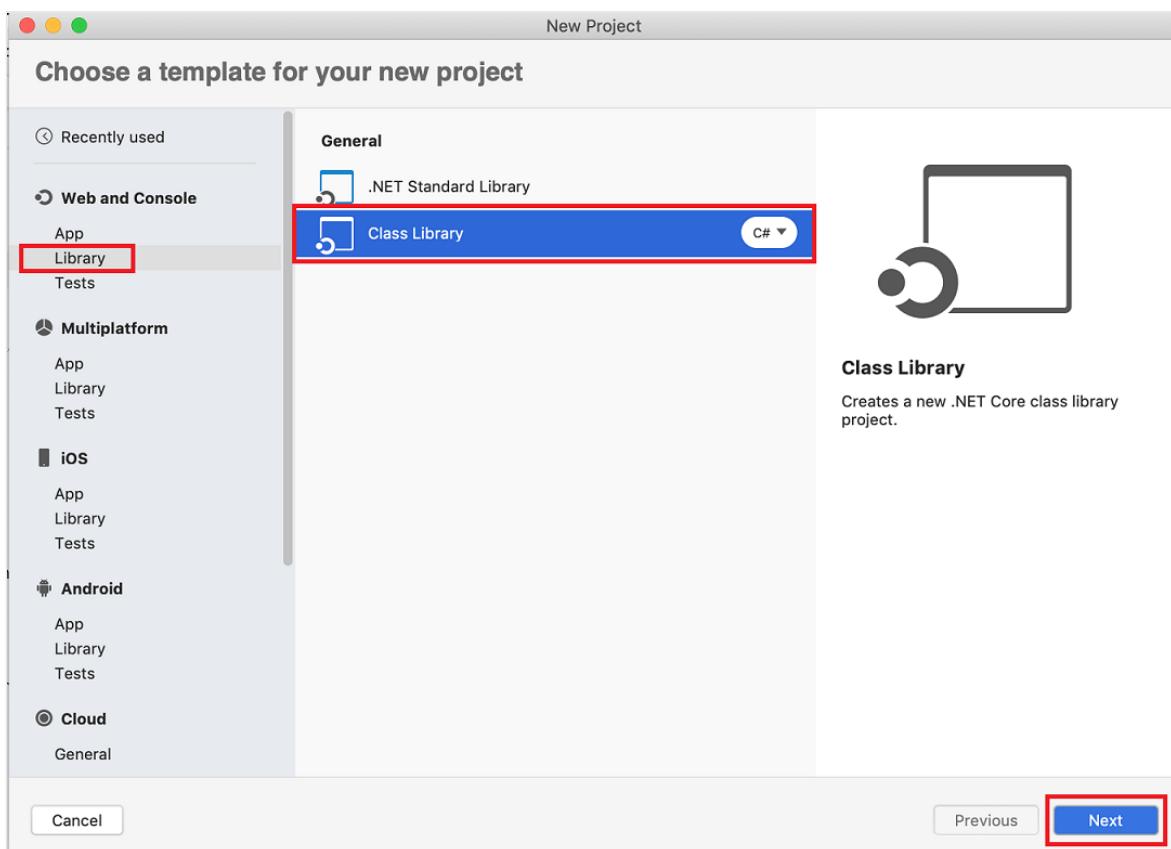
Prerequisites

- [Install Visual Studio for Mac version 8.8 or later](#). Select the option to install .NET Core. Installing Xamarin is optional for .NET development. For more information, see the following resources:
 - [Tutorial: Install Visual Studio for Mac](#).
 - [Supported macOS versions](#).
 - [.NET versions supported by Visual Studio for Mac](#).

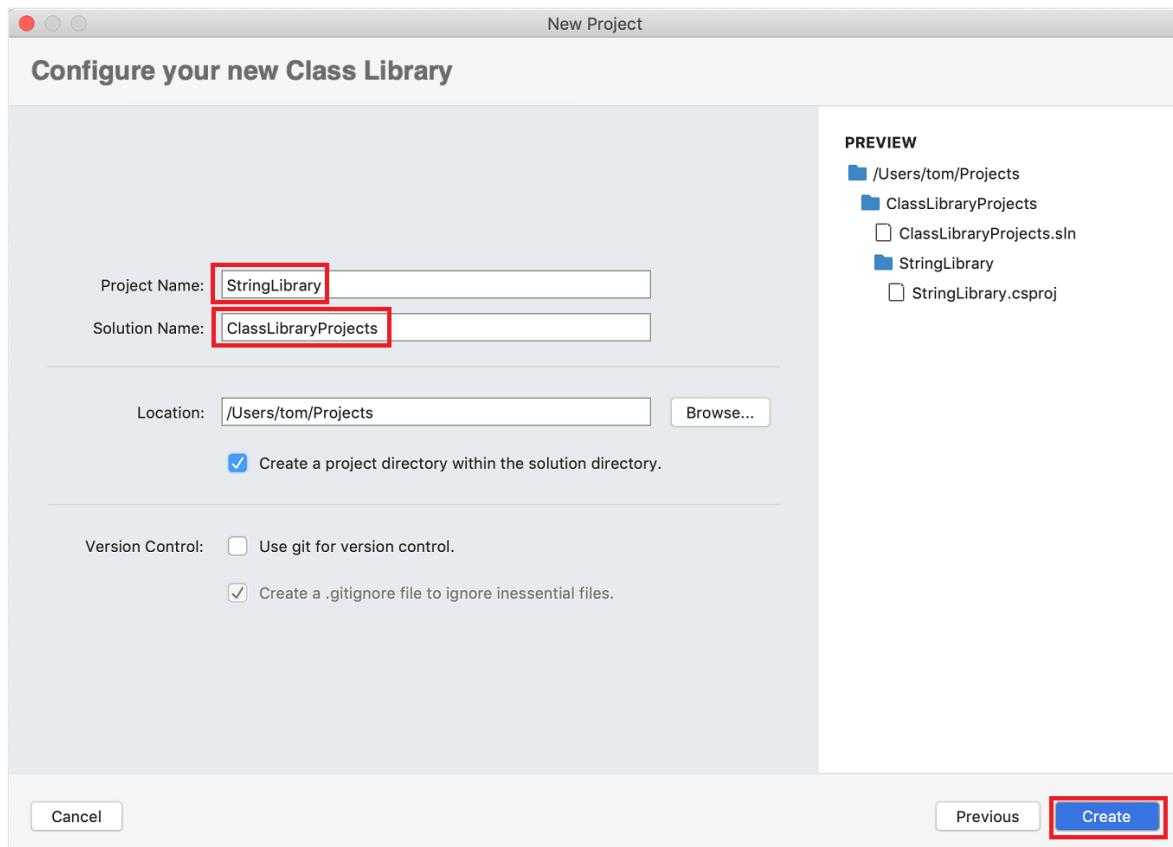
Create a solution with a class library project

A Visual Studio solution serves as a container for one or more projects. Create a solution and a class library project in the solution. You'll add additional, related projects to the same solution later.

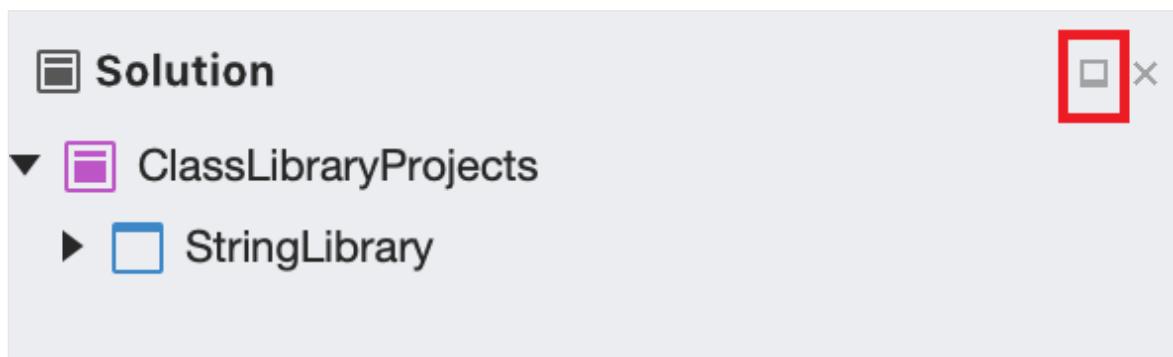
1. Start Visual Studio for Mac.
2. In the start window, select **New Project**.
3. In the **Choose a template for your new project** dialog select **Web and Console > Library > Class Library**, and then select **Next**.



4. In the **Configure your new Class Library** dialog, choose **.NET 5.0**, and select **Next**.
5. Name the project "StringLibrary" and the solution "ClassLibraryProjects". Leave **Create a project directory within the solution directory** selected. Select **Create**.



6. From the main menu, select **View > Solution**, and select the dock icon to keep the pad open.



7. In the **Solution** pad, expand the `StringLibrary` node to reveal the class file provided by the template, `Class1.cs`. `ctrl`-click the file, select **Rename** from the context menu, and rename the file to `StringLibrary.cs`. Open the file and replace the contents with the following code:

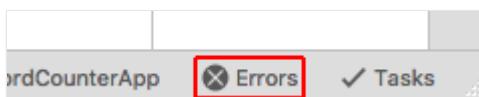
```
C#  
  
using System;  
  
namespace UtilityLibraries  
{  
    public static class StringLibrary  
    {  
        public static bool StartsWithUpper(this string str)  
        {
```

```
        if (string.IsNullOrWhiteSpace(str))
            return false;

        char ch = str[0];
        return char.ToUpper(ch);
    }
}
```

8. Press **⌘S** (**command** + **S**) to save the file.

9. Select **Errors** in the margin at the bottom of the IDE window to open the **Errors** panel. Select the **Build Output** button.



10. Select **Build > Build All** from the menu.

The solution builds. The build output panel shows that the build is successful.

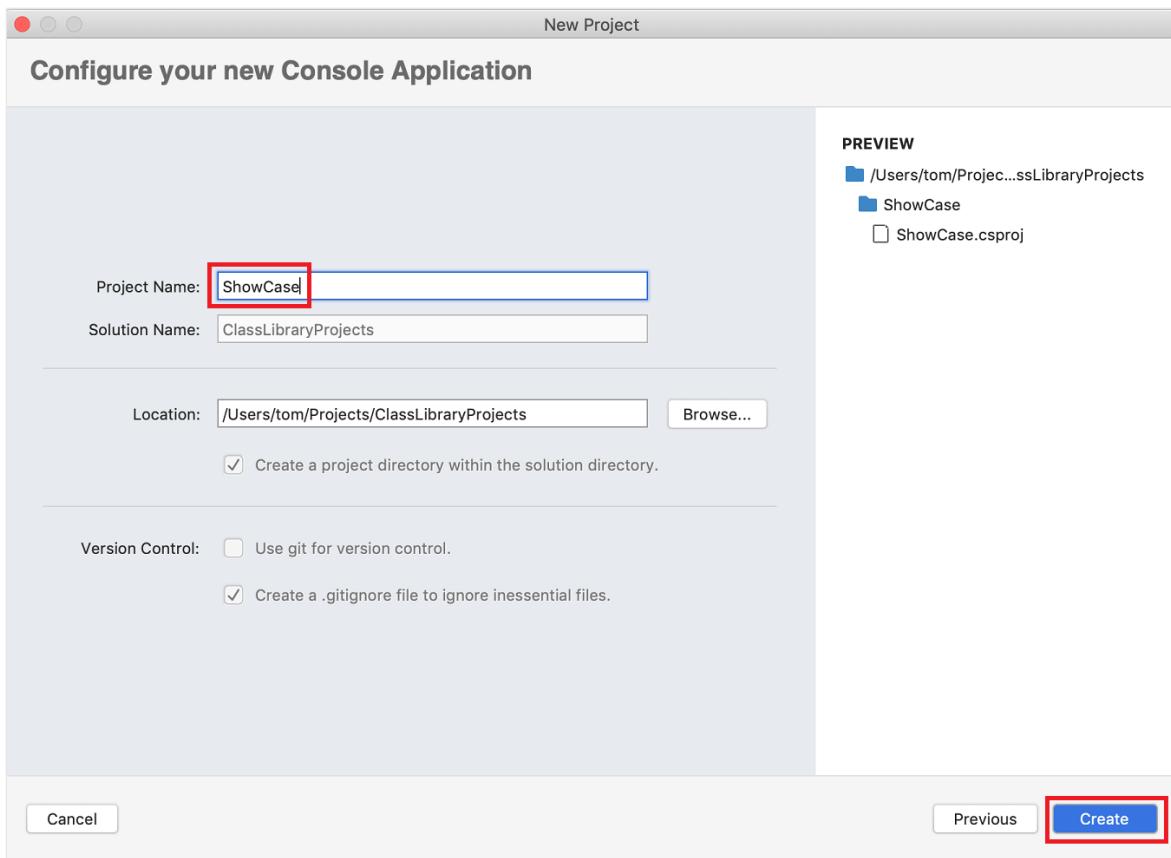
```
Build succeeded.
0 Warning(s)
0 Error(s)

Time Elapsed 00:00:01.50
=========
Build: 1 succeeded, 0 failed, 0 up-to-date, 0 skipped ======
Build successful.
```

Add a console app to the solution

Add a console application that uses the class library. The app will prompt the user to enter a string and report whether the string begins with an uppercase character.

1. In the **Solution** pad, **ctrl**-click the **ClassLibraryProjects** solution. Add a new **Console Application** project by selecting the template from the **Web and Console > App** templates, and select **Next**.
2. Select **.NET 5.0** as the **Target Framework** and select **Next**.
3. Name the project **ShowCase**. Select **Create** to create the project in the solution.



4. Open the *Program.cs* file. Replace the code with the following code:

```
C#  
  
using System;  
using UtilityLibraries;  
  
class Program  
{  
    static void Main(string[] args)  
    {  
        int row = 0;  
  
        do  
        {  
            if (row == 0 || row >= 25)  
                ResetConsole();  
  
            string? input = Console.ReadLine();  
            if (string.IsNullOrEmpty(input)) break;  
            Console.WriteLine($"Input: {input} {"Begins with uppercase?"  
,30}: " +  
                $"{(input.StartsWithUpper() ? "Yes" :  
                    "No")}{Environment.NewLine}");  
            row += 3;  
        } while (true);  
        return;  
  
        // Declare a ResetConsole local method  
        void ResetConsole()  
    }  
}
```

```
        {
            if (row > 0)
            {
                Console.WriteLine("Press any key to continue...");  
                Console.ReadKey();
            }
            Console.Clear();
            Console.WriteLine($"{Environment.NewLine}Press <Enter> only  
to exit; otherwise, enter a string and press <Enter>:  
{Environment.NewLine}");
            row = 3;
        }
    }
}
```

The program prompts the user to enter a string. It indicates whether the string starts with an uppercase character. If the user presses the `enter` key without entering a string, the application ends, and the console window closes.

The code uses the `row` variable to maintain a count of the number of rows of data written to the console window. Whenever it's greater than or equal to 25, the code clears the console window and displays a message to the user.

Add a project reference

Initially, the new console app project doesn't have access to the class library. To allow it to call methods in the class library, create a project reference to the class library project.

1. In the **Solutions** pad, `ctrl`-click the **Dependencies** node of the new **ShowCase** project. In the context menu, select **Add Reference**.
2. In the **References** dialog, select **StringLibrary** and select **OK**.

Run the app

1. `ctrl`-click the **ShowCase** project and select **Run project** from the context menu.
2. Try out the program by entering strings and pressing `enter`, then press `enter` to exit.

Terminal – ShowCase

```
Press <Enter> only to exit; otherwise, enter a string and press <Enter>:
```

```
Begins with uppercase
```

```
Input: Begins with uppercase
```

```
Begins with uppercase? : Yes
```

```
begins with lowercase
```

```
Input: begins with lowercase
```

```
Begins with uppercase? : No
```

Additional resources

- [Develop libraries with the .NET CLI](#)
- [Visual Studio 2019 for Mac Release Notes](#)
- [.NET Standard versions and the platforms they support.](#)

Next steps

In this tutorial, you created a solution and a library project, and added a console app project that uses the library. In the next tutorial, you add a unit test project to the solution.

[Test a .NET class library using Visual Studio for Mac](#)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

Test a .NET class library using Visual Studio

Article • 09/08/2023

ⓘ Important

Microsoft has announced the retirement of Visual Studio for Mac. Visual Studio for Mac will no longer be supported starting August 31, 2024. Alternatives include:

- Visual Studio Code with the [C# Dev Kit](#) and related extensions, such as [.NET MAUI](#) and [Unity](#).
- Visual Studio running on Windows in a VM on Mac.
- Visual Studio running on Windows in a [VM in the Cloud](#).

For more information, see [Visual Studio for Mac retirement announcement](#).

This tutorial shows how to automate unit testing by adding a test project to a solution.

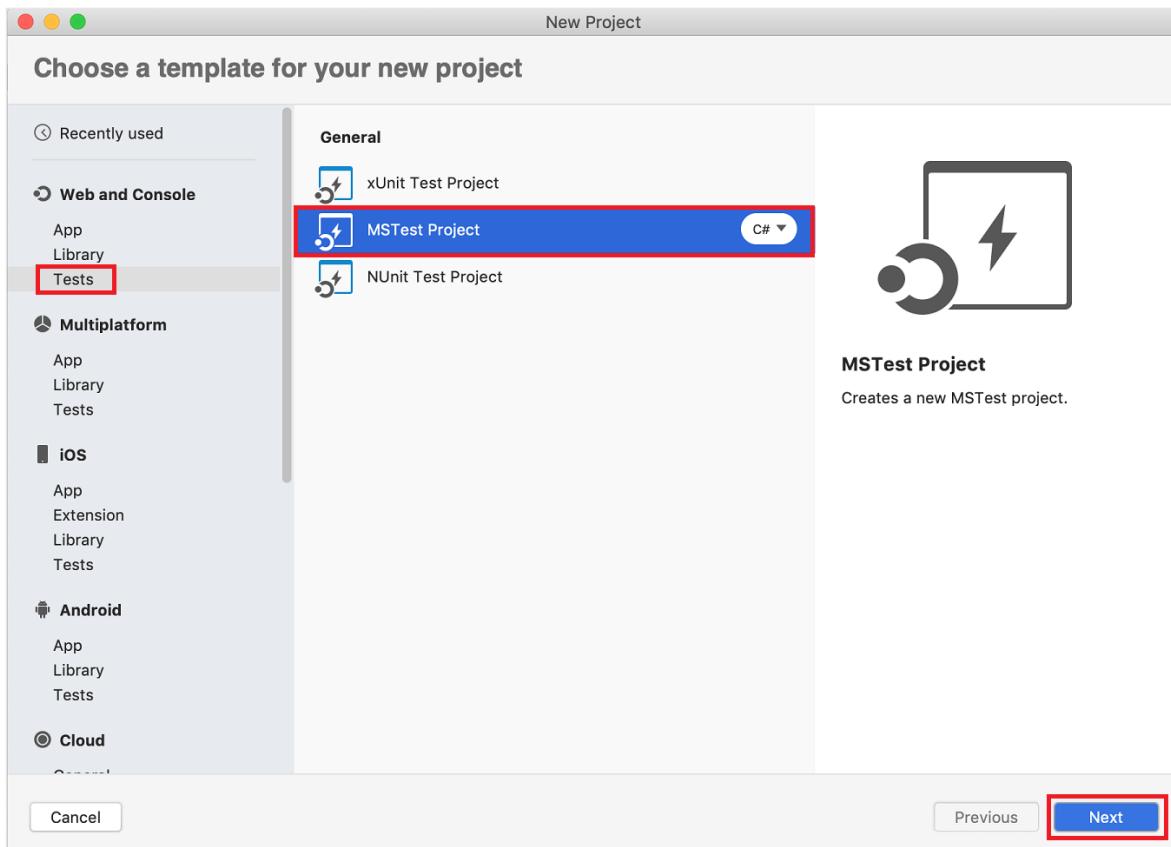
Prerequisites

- This tutorial works with the solution that you create in [Create a .NET class library using Visual Studio for Mac](#).

Create a unit test project

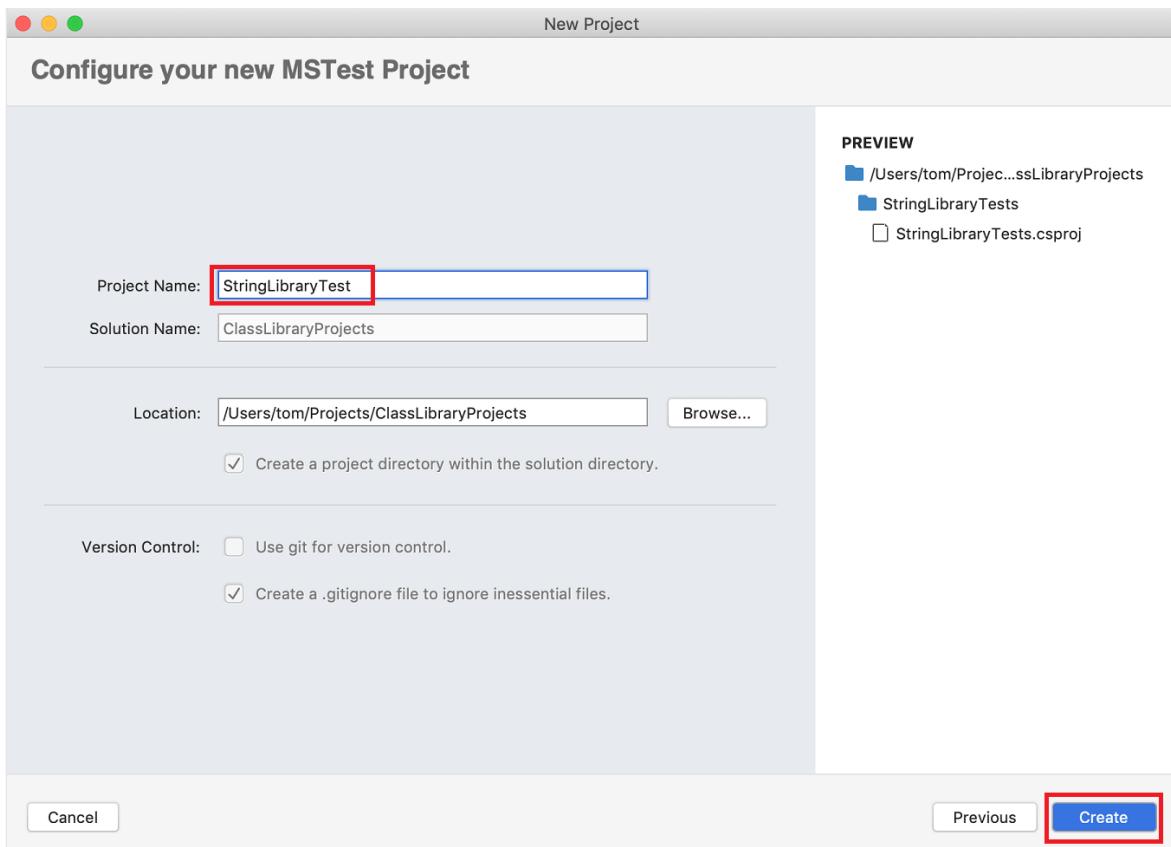
Unit tests provide automated software testing during your development and publishing. [MSTest](#) is one of three test frameworks you can choose from. The others are [xUnit](#) and [nUnit](#).

1. Start Visual Studio for Mac.
2. Open the `ClassLibraryProjects` solution you created in [Create a .NET class library using Visual Studio for Mac](#).
3. In the **Solution** pad, **ctrl**-click the `ClassLibraryProjects` solution and select **Add > New Project**.
4. In the **New Project** dialog, select **Tests** from the **Web and Console** node. Select the **MSTest Project** followed by **Next**.



5. Select .NET 5.0 as the Target Framework and select **Next**.

6. Name the new project "StringLibraryTest" and select **Create**.



Visual Studio creates a class file with the following code:

C#

```
using Microsoft.VisualStudio.TestTools.UnitTesting;

namespace StringLibraryTest
{
    [TestClass]
    public class UnitTest1
    {
        [TestMethod]
        public void TestMethod1()
        {
        }
    }
}
```

The source code created by the unit test template does the following:

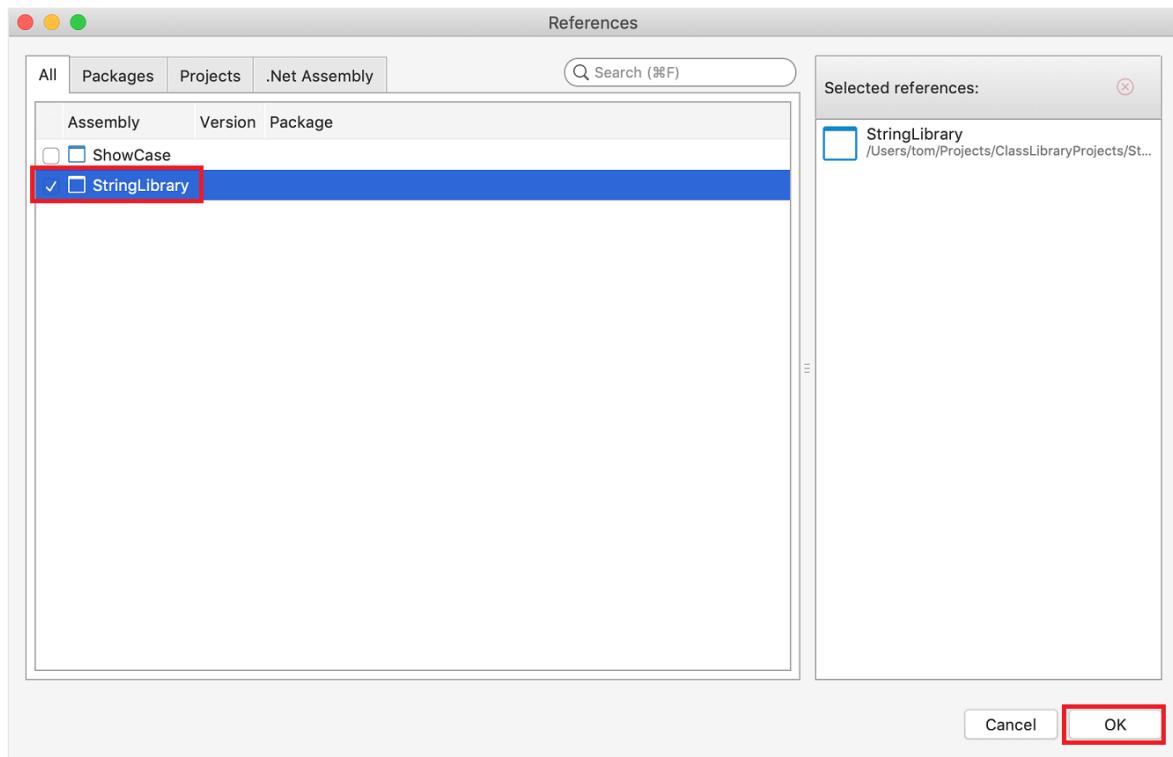
- It imports the `Microsoft.VisualStudio.TestTools.UnitTesting` namespace, which contains the types used for unit testing.
- It applies the `TestClassAttribute` attribute to the `UnitTest1` class.
- It applies the `TestMethodAttribute` attribute to `TestMethod1`.

Each method tagged with `[TestMethod]` in a test class tagged with `[TestClass]` is executed automatically when the unit test is run.

Add a project reference

For the test project to work with the `StringLibrary` class, add a reference to the `StringLibrary` project.

1. In the **Solution** pad, **ctrl**-click **Dependencies** under `StringLibraryTest`. Select **Add Reference** from the context menu.
2. In the **References** dialog, select the `StringLibrary` project. Select **OK**.



Add and run unit test methods

When Visual Studio runs a unit test, it executes each method that is marked with the [TestMethodAttribute](#) attribute in a class that is marked with the [TestClassAttribute](#) attribute. A test method ends when the first failure is found or when all tests contained in the method have succeeded.

The most common tests call members of the [Assert](#) class. Many assert methods include at least two parameters, one of which is the expected test result and the other of which is the actual test result. Some of the [Assert](#) class's most frequently called methods are shown in the following table:

| Assert methods | Function |
|-------------------------------|--|
| <code>Assert.AreEqual</code> | Verifies that two values or objects are equal. The assert fails if the values or objects aren't equal. |
| <code>Assert.AreSame</code> | Verifies that two object variables refer to the same object. The assert fails if the variables refer to different objects. |
| <code>Assert.IsFalse</code> | Verifies that a condition is <code>false</code> . The assert fails if the condition is <code>true</code> . |
| <code>Assert.IsNotNull</code> | Verifies that an object isn't <code>null</code> . The assert fails if the object is <code>null</code> . |

You can also use the [Assert.ThrowsException](#) method in a test method to indicate the type of exception it's expected to throw. The test fails if the specified exception isn't thrown.

In testing the `StringLibrary.StartsWithUpper` method, you want to provide a number of strings that begin with an uppercase character. You expect the method to return `true` in these cases, so you can call the `Assert.IsTrue` method. Similarly, you want to provide a number of strings that begin with something other than an uppercase character. You expect the method to return `false` in these cases, so you can call the `Assert.IsFalse` method.

Since your library method handles strings, you also want to make sure that it successfully handles an `empty string (String.Empty)`, a valid string that has no characters and whose `Length` is 0, and a `null` string that hasn't been initialized. You can call `StartsWithUpper` directly as a static method and pass a single `String` argument. Or you can call `StartsWithUpper` as an extension method on a `string` variable assigned to `null`.

You'll define three methods, each of which calls an `Assert` method for each element in a string array. You'll call a method overload that lets you specify an error message to be displayed in case of test failure. The message identifies the string that caused the failure.

To create the test methods:

1. Open the `UnitTest1.cs` file and replace the code with the following code:

```
C#  
  
using Microsoft.VisualStudio.TestTools.UnitTesting;  
using UtilityLibraries;  
  
namespace StringLibraryTest  
{  
    [TestClass]  
    public class UnitTest1  
    {  
        [TestMethod]  
        public void TestStartsWithUpper()  
        {  
            // Tests that we expect to return true.  
            string[] words = { "Alphabet", "Zebra", "ABC", "Aθήνα",  
"Москва" };  
            foreach (var word in words)  
            {  
                bool result = word.StartsWithUpper();  
                Assert.IsTrue(result,  
                    string.Format("Expected for '{0}': true; Actual:  
{1}",  
                                word, result));  
            }  
        }  
    }  
}
```

```

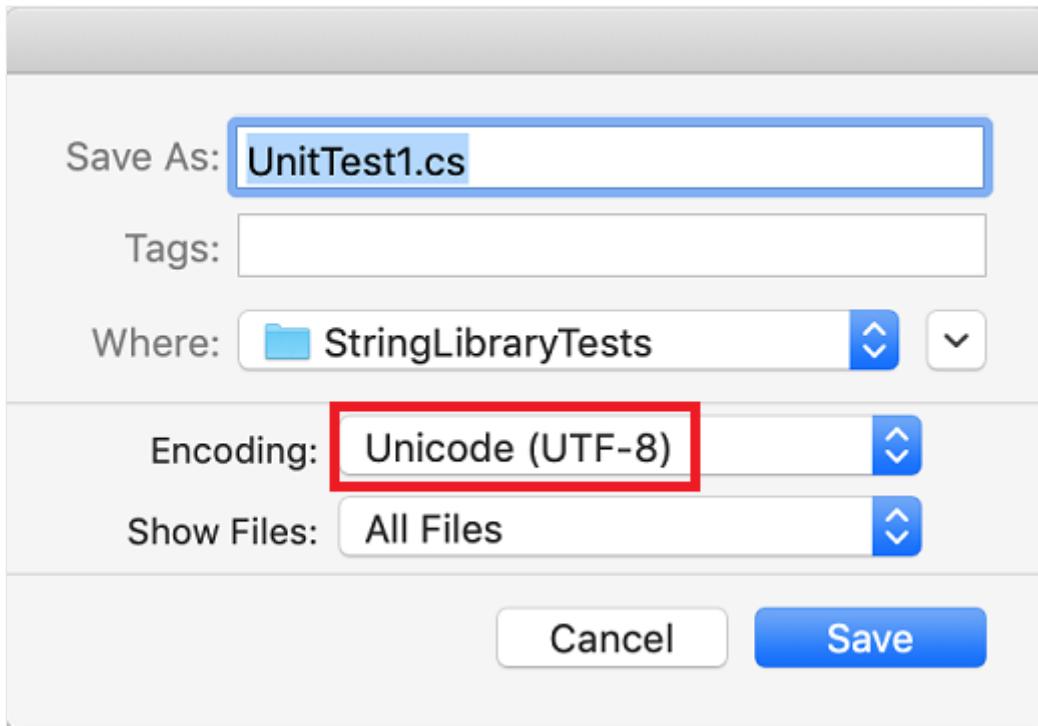
[TestMethod]
public void TestDoesNotStartWithUpper()
{
    // Tests that we expect to return false.
    string[] words = { "alphabet", "zebra", "abc",
"αυτοκινητοβιομηχανία", "государство",
"1234", ".", ";", " " };
    foreach (var word in words)
    {
        bool result = word.StartsWithUpper();
        Assert.IsFalse(result,
            string.Format("Expected for '{0}': false;
Actual: {1}",
                           word, result));
    }
}

[TestMethod]
public void DirectCallWithNullOrEmpty()
{
    // Tests that we expect to return false.
    string?[] words = { string.Empty, null };
    foreach (var word in words)
    {
        bool result = StringLibrary.StartsWithUpper(word);
        Assert.IsFalse(result,
            string.Format("Expected for '{0}': false;
Actual: {1}",
                           word == null ? "<null>" : word,
result));
    }
}

```

The test of uppercase characters in the `TestStartsWithUpper` method includes the Greek capital letter alpha (U+0391) and the Cyrillic capital letter EM (U+041C). The test of lowercase characters in the `TestDoesNotStartWithUpper` method includes the Greek small letter alpha (U+03B1) and the Cyrillic small letter Ghe (U+0433).

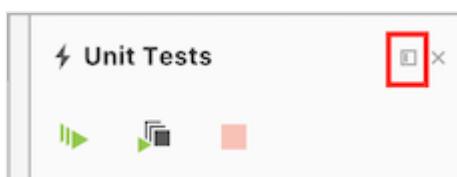
2. On the menu bar, select **File > Save As**. In the dialog, make sure that **Encoding** is set to **Unicode (UTF-8)**.



3. When you're asked if you want to replace the existing file, select **Replace**.

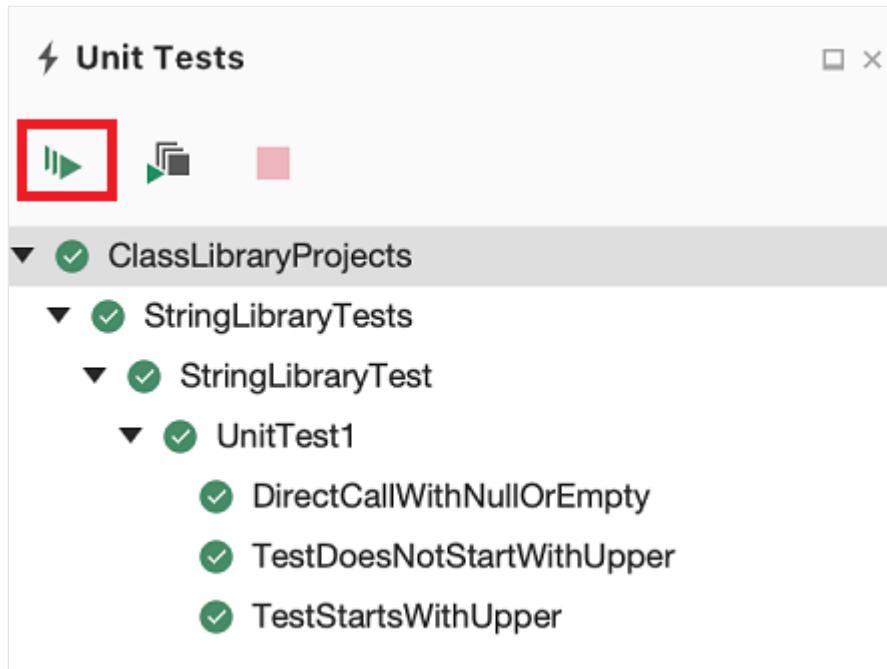
If you fail to save your source code as a UTF8-encoded file, Visual Studio may save it as an ASCII file. When that happens, the runtime doesn't accurately decode the UTF8 characters outside of the ASCII range, and the test results won't be correct.

4. Open the **Unit Tests** panel on the right side of the screen. Select **View > Tests** from the menu.
5. Click the **Dock** icon to keep the panel open.



6. Click the **Run All** button.

All tests pass.



Handle test failures

If you're doing test-driven development (TDD), you write tests first and they fail the first time you run them. Then you add code to the app that makes the test succeed. For this tutorial, you created the test after writing the app code that it validates, so you haven't seen the test fail. To validate that a test fails when you expect it to fail, add an invalid value to the test input.

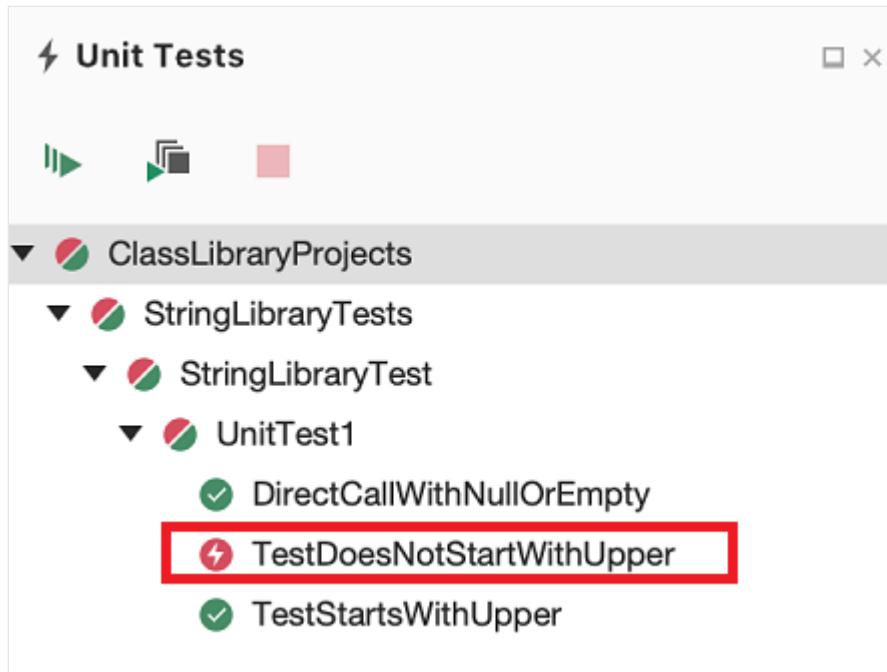
1. Modify the `words` array in the `TestDoesNotStartWithUpper` method to include the string "Error". You don't need to save the file because Visual Studio automatically saves open files when a solution is built to run tests.

C#

```
string[] words = { "alphabet", "Error", "zebra", "abc",
    "αυτοκινητοβιομηχανία", "государство",
    "1234", ".", ";", " " };
```

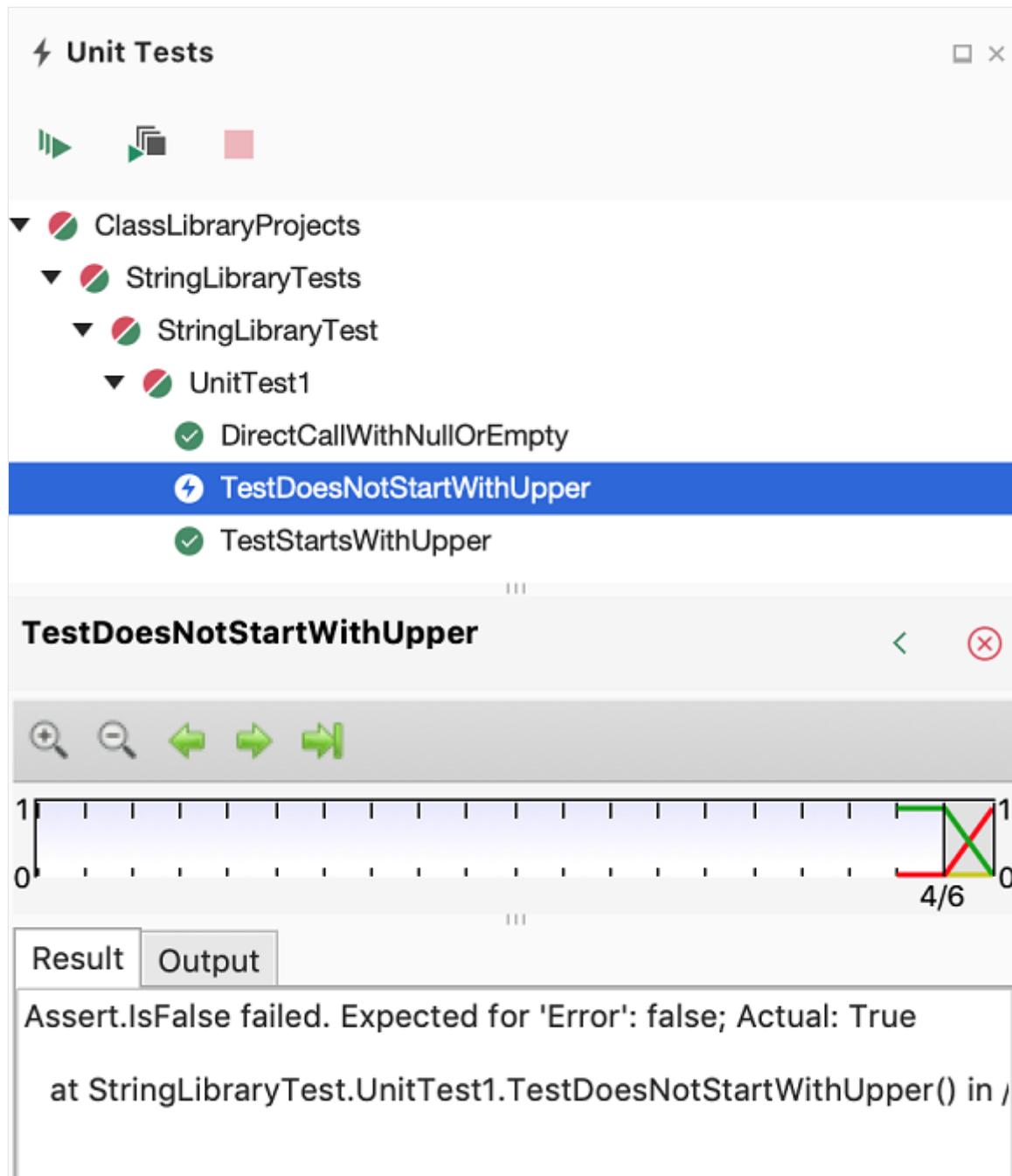
2. Run the tests again.

This time, the **Test Explorer** window indicates that two tests succeeded and one failed.



3. `ctrl`-click the failed test, `TestDoesNotStartWithUpper`, and select **Show Results Pad** from the context menu.

The **Results** pad displays the message produced by the assert: "Assert.IsFalse failed. Expected for 'Error': false; actual: True". Because of the failure, no strings in the array after "Error" were tested.



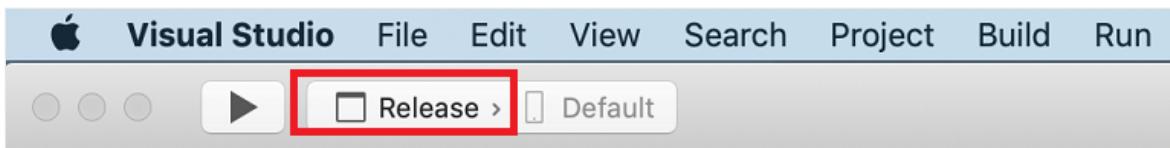
4. Remove the string "Error" that you added in step 1. Rerun the test and the tests pass.

Test the Release version of the library

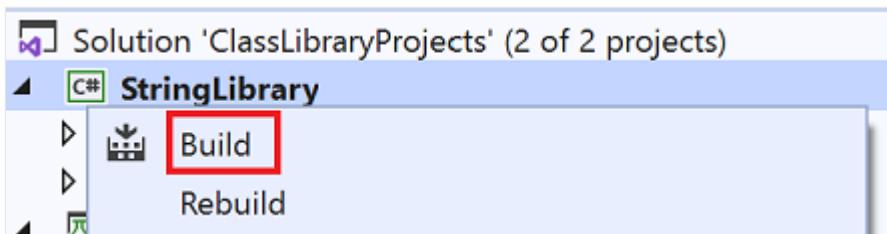
Now that the tests have all passed when running the Debug build of the library, run the tests an additional time against the Release build of the library. A number of factors, including compiler optimizations, can sometimes produce different behavior between Debug and Release builds.

To test the Release build:

1. In the Visual Studio toolbar, change the build configuration from **Debug** to **Release**.



2. In the Solution pad, ctrl -click the **StringLibrary** project and select **Build** from the context menu to recompile the library.



3. Run the unit tests again.

The tests pass.

Debug tests

If you're using Visual Studio for Mac as your IDE, you can use the same process shown in [Tutorial: Debug a .NET console application using Visual Studio for Mac](#) to debug code using your unit test project. Instead of starting the *ShowCase* app project, ctrl -click the **StringLibraryTests** project, and select **Start Debugging Project** from the context menu.

Visual Studio starts the test project with the debugger attached. Execution will stop at any breakpoint you've added to the test project or the underlying library code.

Additional resources

- [Unit testing in .NET](#)

Next steps

In this tutorial, you unit tested a class library. You can make the library available to others by publishing it to [NuGet](#) as a package. To learn how, follow a NuGet tutorial:

[Create and publish a package \(dotnet CLI\)](#)

If you publish a library as a NuGet package, others can install and use it. To learn how, follow a NuGet tutorial:

[Install and use a package in Visual Studio for Mac](#)

A library doesn't have to be distributed as a package. It can be bundled with a console app that uses it. To learn how to publish a console app, see the earlier tutorial in this series:

Publish a .NET console application using Visual Studio for Mac

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

Quickstart: Install and use a package in Visual Studio for Mac

Article • 09/03/2019

NuGet packages contain reusable code that other developers make available to you for use in your projects. See [What is NuGet?](#) for background. Packages are installed into a Visual Studio for Mac project using the NuGet Package Manager. This article demonstrates the process using the popular [Newtonsoft.Json](#) package and a .NET Core console project. The same process applies to any other Xamarin or .NET Core project.

Once installed, refer to the package in code with `using <namespace>` where `<namespace>` is specific to the package you're using. Once the reference is made, you can call the package through its API.

Tip

Start with nuget.org: Browsing *nuget.org* is how .NET developers typically find components they can reuse in their own applications. You can search *nuget.org* directly or find and install packages within Visual Studio as shown in this article. For general information, see [Find and evaluate NuGet packages](#).

Prerequisites

- Visual Studio 2019 for Mac.

You can install the 2019 Community edition for free from [visualstudio.com](#) or use the Professional or Enterprise editions.

If you're using Visual Studio on Windows, see [Install and use a package in Visual Studio \(Windows Only\)](#).

Create a project

NuGet packages can be installed into any .NET project, provided that the package supports the same target framework as the project.

For this walkthrough, use a simple .NET Core Console app. Create a project in Visual Studio for Mac using **File > New Solution...**, select the **.NET Core > App > Console**

Application template. Click **Next**. Accept the default values for **Target Framework** when prompted.

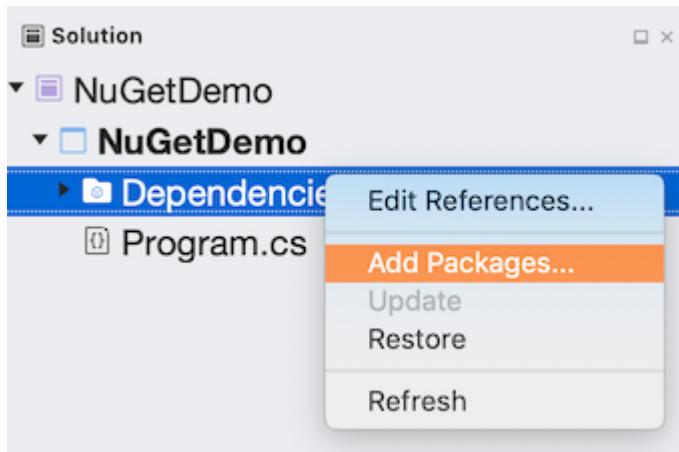
Visual Studio creates the project, which opens in Solution Explorer.

Add the Newtonsoft.Json NuGet package

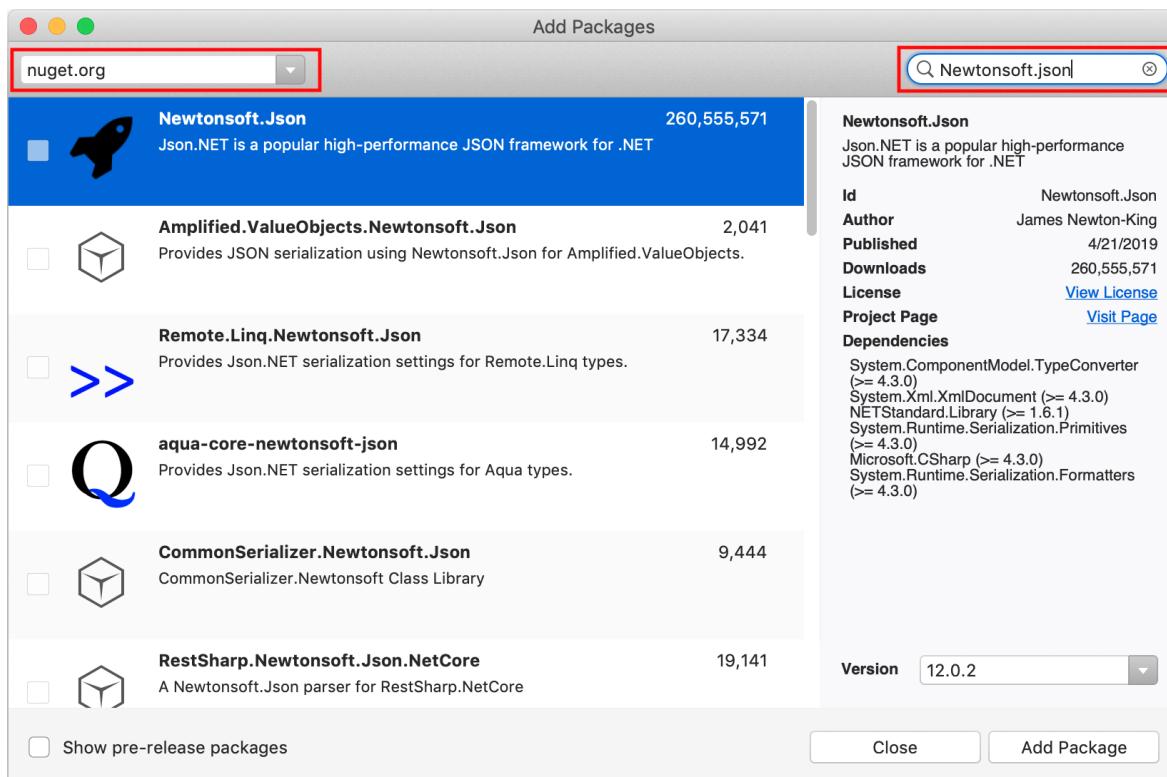
To install the package, you use the NuGet Package Manager. When you install a package, NuGet records the dependency in either your project file or a `packages.config` file (depending on the project format). For more information, see [Package consumption overview and workflow](#).

NuGet Package Manager

1. In Solution Explorer, right-click **Dependencies** and choose **Add Packages...**



2. Choose "nuget.org" as the **Package source** in the top left corner of the dialog, and search for **Newtonsoft.Json**, select that package in the list, and select **Add Packages...**:



If you want more information on the NuGet Package Manager, see [Install and manage packages using Visual Studio for Mac](#).

Use the Newtonsoft.Json API in the app

With the Newtonsoft.Json package in the project, you can call its `JsonConvert.SerializeObject` method to convert an object to a human-readable string.

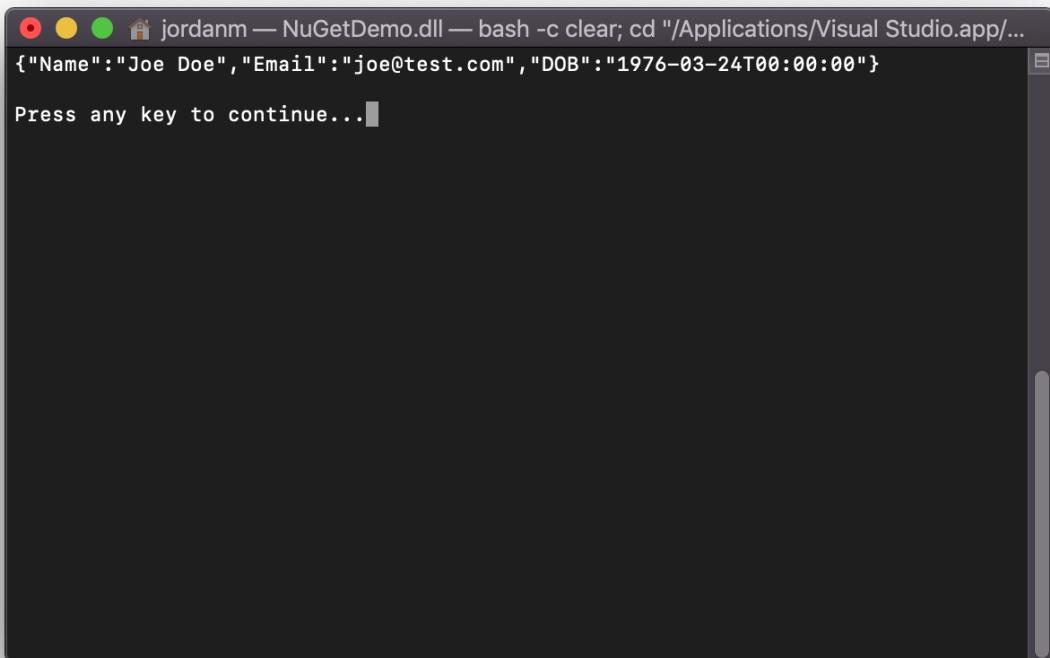
1. Open the `Program.cs` file (located in the Solution Pad) and replace the file contents with the following code:

```
C#  
  
using System;  
using Newtonsoft.Json;  
  
namespace NuGetDemo  
{  
    public class Account  
    {  
        public string Name { get; set; }  
        public string Email { get; set; }  
        public DateTime DOB { get; set; }  
    }  
  
    class Program  
    {  
        static void Main(string[] args)  
        {  
        }  
    }  
}
```

```
Account account = new Account()
{
    Name = "Joe Doe",
    Email = "joe@test.com",
    DOB = new DateTime(1976, 3, 24)
};
string json = JsonConvert.SerializeObject(account);
Console.WriteLine(json);
}
```

2. Build and run the app by selecting **Run > Start Debugging**:

3. Once the app runs, you'll see the serialized JSON output appear in the console:



```
jordanm — NuGetDemo.dll — bash -c clear; cd "/Applications/Visual Studio.app/..."; ./NuGetDemo
{"Name": "Joe Doe", "Email": "joe@test.com", "DOB": "1976-03-24T00:00:00"}
Press any key to continue...
```

Next steps

Congratulations on installing and using your first NuGet package!

[Install and manage packages using Visual Studio for Mac](#)

To explore more that NuGet has to offer, select the links below.

- [Overview and workflow of package consumption](#)
- [Package references in project files](#)

Learn .NET and the .NET SDK tools by exploring these tutorials

Article • 09/08/2023

The following tutorials show how to develop console apps and libraries for .NET Core, .NET 5, and later versions. For other types of applications, see [Tutorials for getting started with .NET](#).

Use Visual Studio

- [Create a console app](#)
- [Debug an app](#)
- [Publish an app](#)
- [Create a class library](#)
- [Unit test a class library](#)
- [Install and use a package](#)
- [Create and publish a package](#)
- [Create an F# console app](#)

Use Visual Studio Code

Choose these tutorials if you want to use Visual Studio Code or some other code editor. All of them use the CLI for .NET Core development tasks, so all except the debugging tutorial can be used with any code editor.

- [Create a console app](#)
- [Debug an app](#)
- [Publish an app](#)
- [Create a class library](#)
- [Unit test a class library](#)
- [Install and use a package](#)
- [Create and publish a package](#)
- [Create an F# console app](#)

Advanced articles

- [How to create libraries](#)
- [Unit test an app with xUnit](#)

- Unit test using C#/VB/F# with NUnit/xUnit/MSTest
- Live unit test with Visual Studio
- Create templates for the CLI
- Create and use tools for the CLI
- Create an app with plugins

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

What's new in .NET 8

Article • 11/14/2023

.NET 8 is the successor to [.NET 7](#). It will be [supported for three years](#) as a long-term support (LTS) release. You can [download .NET 8 here](#).

.NET Aspire

.NET Aspire is an opinionated, cloud ready stack for building observable, production ready, distributed applications. .NET Aspire is delivered through a collection of NuGet packages that handle specific cloud-native concerns, and is available in preview for .NET 8. For more information, see [.NET Aspire \(Preview\)](#).

ASP.NET Core

For information about what's new in ASP.NET Core, see [What's new in ASP.NET Core 8.0](#).

Core .NET libraries

This section contains the following subtopics:

- [Serialization](#)
- [Time abstraction](#)
- [UTF8 improvements](#)
- [Methods for working with randomness](#)
- [Performance-focused types](#)
- [System.Numerics and System.Runtime.Intrinsics](#)
- [Data validation](#)
- [Metrics](#)
- [Cryptography](#)
- [Networking](#)
- [Stream-based ZipFile methods](#)

Serialization

Many improvements have been made to [System.Text.Json](#) serialization and deserialization functionality in .NET 8. For example, you can [customize handling of members that aren't in the JSON payload](#).

The following sections describe other serialization improvements:

- Built-in support for additional types
- Source generator
- Interface hierarchies
- Naming policies
- Read-only properties
- Disable reflection-based default
- New `JsonNode` API methods
- Non-public members
- Streaming deserialization APIs
- `WithAddedModifier` extension method
- New `JsonContent.Create` overloads
- Freeze a `JsonSerializerOptions` instance

For more information about JSON serialization in general, see [JSON serialization and deserialization in .NET](#).

Built-in support for additional types

The serializer has built-in support for the following additional types.

- `Half`, `Int128`, and `UInt128` numeric types.

```
C#  
  
Console.WriteLine(JsonSerializer.Serialize(  
    [ Half.MaxValue, Int128.MaxValue, UInt128.MaxValue ]  
));  
//  
[65500,170141183460469231731687303715884105727,340282366920938463463374  
607431768211455]
```

- `Memory<T>` and `ReadOnlyMemory<T>` values. `byte` values are serialized to Base64 strings, and other types to JSON arrays.

```
C#  
  
JsonSerializer.Serialize<ReadOnlyMemory<byte>>(new byte[] { 1, 2, 3 });  
// "AQID"  
JsonSerializer.Serialize<Memory<int>>(new int[] { 1, 2, 3 }); //  
[1,2,3]
```

Source generator

.NET 8 includes enhancements of the `System.Text.Json` [source generator](#) that are aimed at making the [Native AOT](#) experience on par with the [reflection-based serializer](#). For example:

- The source generator now supports serializing types with `required` and `init` properties. These were both already supported in reflection-based serialization.
- Improved formatting of source-generated code.
- `JsonSourceGenerationOptionsAttribute` feature parity with `JsonSerializerOptions`. For more information, see [Specify options \(source generation\)](#).
- Additional diagnostics (such as [SYSLIB1034](#) and [SYSLIB1039](#)).
- Don't include types of ignored or inaccessible properties.
- Support for nesting `JsonSerializerContext` declarations within arbitrary type kinds.
- Support for compiler-generated or *unspeakable* types in weakly typed source generation scenarios. Since compiler-generated types can't be explicitly specified by the source generator, `System.Text.Json` now performs nearest-ancestor resolution at run time. This resolution determines the most appropriate supertype with which to serialize the value.
- New converter type `JsonStringEnumConverter<TEnum>`. The existing `JsonStringEnumConverter` class isn't supported in Native AOT. You can annotate your enum types as follows:

C#

```
[JsonConverter(typeof(JsonStringEnumConverter<MyEnum>))]
public enum MyEnum { Value1, Value2, Value3 }

[JsonSerializable(typeof(MyEnum))]
public partial class MyContext : JsonSerializerContext { }
```

For more information, see [Serialize enum fields as strings](#).

- New `JsonConverter.Type` property lets you look up the type of a non-generic `JsonConverter` instance:

C#

```
Dictionary<Type, JsonConverter>
CreateDictionary(IEnumerable<JsonConverter> converters)
    => converters.Where(converter => converter.Type != null)
        .ToDictionary(converter => converter.Type!);
```

The property is nullable since it returns `null` for `JsonConverterFactory` instances and `typeof(T)` for `JsonConverter<T>` instances.

Chain source generators

The `JsonSerializerOptions` class includes a new `TypeInfoResolverChain` property that complements the existing `TypeInfoResolver` property. These properties are used in contract customization for chaining source generators. The addition of the new property means that you don't have to specify all chained components at one call site—they can be added after the fact. `TypeInfoResolverChain` also lets you introspect the chain or remove components from it. For more information, see [Combine source generators](#).

In addition, `JsonSerializerOptions.AddContext<TContext>()` is now obsolete. It's been superseded by the `TypeInfoResolver` and `TypeInfoResolverChain` properties. For more information, see [SYSLIB0049](#).

Interface hierarchies

.NET 8 adds support for serializing properties from interface hierarchies.

The following code shows an example where the properties from both the immediately implemented interface and its base interface are serialized.

C#

```
IDerived value = new DerivedImplement { Base = 0, Derived = 1 };
JsonSerializer.Serialize(value);
// {"Base":0,"Derived":1}

public interface IBase
{
    public int Base { get; set; }
}

public interface IDerived : IBase
{
    public int Derived { get; set; }
}

public class DerivedImplement : IDerived
{
```

```
    public int Base { get; set; }
    public int Derived { get; set; }
}
```

Naming policies

[JsonNamingPolicy](#) includes new naming policies for `snake_case` (with an underscore) and `kebab-case` (with a hyphen) property name conversions. Use these policies similarly to the existing [JsonNamingPolicy.CamelCase](#) policy:

C#

```
var options = new JsonSerializerOptions
{
    PropertyNamingPolicy = JsonNamingPolicy.SnakeCaseLower
};
JsonSerializer.Serialize(new {PropertyName = "value"}, options);
// { "property_name" : "value" }
```

For more information, see [Use a built-in naming policy](#).

Read-only properties

You can now deserialize onto read-only fields or properties (that is, those that don't have a `set` accessor).

To opt into this support globally, set a new option, [PreferredObjectCreationHandling](#), to [JsonObjectCreationHandling.Populate](#). If compatibility is a concern, you can also enable the functionality more granularly by placing the

`[JsonObjectCreationHandling(JsonObjectCreationHandling.Populate)]` attribute on specific types whose properties are to be populated, or on individual properties.

For example, consider the following code that deserializes into a `CustomerInfo` type that has two read-only properties.

C#

```
using System.Text.Json;

CustomerInfo customer =
    JsonSerializer.Deserialize<CustomerInfo>("""
        {"Names": ["John Doe"], "Company": {"Name": "Contoso"}}
    """)!;

Console.WriteLine(JsonSerializer.Serialize(customer));
```

```
class CompanyInfo
{
    public required string Name { get; set; }
    public string? PhoneNumber { get; set; }
}

[JsonObjectCreationHandling(JsonObjectCreationHandling.Populate)]
class CustomerInfo
{
    // Both of these properties are read-only.
    public List<string> Names { get; } = new();
    public CompanyInfo Company { get; } = new()
    {
        Name = "N/A", PhoneNumber = "N/A"
    };
}
```

Prior to .NET 8, the input values were ignored and the `Names` and `Company` properties retained their default values.

Output

```
{"Names":[], "Company": {"Name": "N/A", "PhoneNumber": "N/A"}}
```

Now, the input values are used to populate the read-only properties during deserialization.

Output

```
{"Names": ["John Doe"], "Company": {"Name": "Contoso", "PhoneNumber": null}}
```

For more information about the *populate* deserialization behavior, see [Populate initialized properties](#).

Disable reflection-based default

You can now disable using the reflection-based serializer by default. This disablement is useful to avoid accidental rooting of reflection components that aren't even in use, especially in trimmed and Native AOT apps. To disable default reflection-based serialization by requiring that a `JsonSerializerOptions` argument be passed to the `JsonSerializer` serialization and deserialization methods, set the `JsonSerializerIsReflectionEnabledByDefault` MSBuild property to `false` in your project file.

Use the new [IsReflectionEnabledByDefault](#) API to check the value of the feature switch. If you're a library author building on top of [System.Text.Json](#), you can rely on the property to configure your defaults without accidentally rooting reflection components.

For more information, see [Disable reflection defaults](#).

New JsonNode API methods

The [JsonNode](#) and [System.Text.Json.Nodes.JsonArray](#) types include the following new methods.

C#

```
public partial class JsonNode
{
    // Creates a deep clone of the current node and all its descendants.
    public JsonNode DeepClone();

    // Returns true if the two nodes are equivalent JSON representations.
    public static bool DeepEquals(JsonNode? node1, JsonNode? node2);

    // Determines the JsonValueKind of the current node.
    public JsonValueKind GetValueKind(JsonSerializerOptions options = null);

    // If node is the value of a property in the parent
    // object, returns its name.
    // Throws InvalidOperationException otherwise.
    public string GetPropertyName();

    // If node is the element of a parent JsonArray,
    // returns its index.
    // Throws InvalidOperationException otherwise.
    public int GetElementIndex();

    // Replaces this instance with a new value,
    // updating the parent object/array accordingly.
    public void ReplaceWith<T>(T value);

    // Asynchronously parses a stream as UTF-8 encoded data
    // representing a single JSON value into a JsonNode.
    public static Task<JsonNode?> ParseAsync(
        Stream utf8Json,
        JsonNodeOptions? nodeOptions = null,
        JsonDocumentOptions documentOptions = default,
        CancellationToken cancellationToken = default);
}

public partial class JsonArray
{
    // Returns an IEnumerable<T> view of the current array.
}
```

```
    public IEnumerable<T> GetValues<T>();  
}
```

Non-public members

You can opt non-public members into the serialization contract for a given type using [JsonIncludeAttribute](#) and [JsonConstructorAttribute](#) attribute annotations.

C#

```
string json = JsonSerializer.Serialize(new MyPoco(42));  
// {"X":42}  
  
JsonSerializer.Deserialize<MyPoco>(json);  
  
public class MyPoco  
{  
    [JsonConstructor]  
    internal MyPoco(int x) => X = x;  
  
    [JsonInclude]  
    internal int X { get; }  
}
```

For more information, see [Use immutable types and non-public members and accessors](#).

Streaming deserialization APIs

.NET 8 includes new [IAsyncEnumerable<T>](#) streaming deserialization extension methods, for example [GetFromJsonAsAsyncEnumerable](#). Similar methods have existed that return [Task<TResult>](#), for example, [HttpClientJsonExtensions.GetFromJsonAsync](#). The new extension methods invoke streaming APIs and return [IAsyncEnumerable<T>](#).

The following code shows how you might use the new extension methods.

C#

```
const string RequestUri = "https://api.contoso.com/books";  
using var client = new HttpClient();  
IAsyncEnumerable<Book> books = client.GetFromJsonAsAsyncEnumerable<Book>  
(RequestUri);  
  
await foreach (Book book in books)  
{  
    Console.WriteLine($"Read book '{book.title}'");  
}
```

```
public record Book(int id, string title, string author, int publishedYear);
```

WithAddedModifier extension method

The new `WithAddedModifier(IJsonTypeInfoResolver, Action<JsonTypeInfo>)` extension method lets you easily introduce modifications to the serialization contracts of arbitrary `IJsonTypeInfoResolver` instances.

C#

```
var options = new JsonSerializerOptions
{
    TypeInfoResolver = MyContext.Default
        .WithAddedModifier(static typeInfo =>
    {
        foreach (JsonPropertyInfo prop in typeInfo.Properties)
        {
            prop.Name = prop.Name.ToUpperInvariant();
        }
    })
};
```

New JsonContent.Create overloads

You can now create `JsonContent` instances using trim-safe or source-generated contracts. The new methods are:

- `JsonContent.Create(Object, JsonTypeInfo, MediaTypeHeaderValue)`
- `JsonContent.Create<T>(T, JsonTypeInfo<T>, MediaTypeHeaderValue)`

C#

```
var book = new Book(id: 42, "Title", "Author", publishedYear: 2023);
HttpContent content = JsonContent.Create(book, MyContext.Default.Book);

public record Book(int id, string title, string author, int publishedYear);

[JsonSerializable(typeof(Book))]
public partial class MyContext : JsonSerializerContext
{}
```

Freeze a JsonSerializerOptions instance

The following new methods let you control when a [JsonSerializerOptions](#) instance is frozen:

- [JsonSerializerOptions.MakeReadOnly\(\)](#)

This overload is designed to be trim-safe and will therefore throw an exception in cases where the options instance hasn't been configured with a resolver.

- [JsonSerializerOptions.MakeReadOnly\(Boolean\)](#)

If you pass `true` to this overload, it populates the options instance with the default reflection resolver if one is missing. This method is marked `RequiresUnreferencedCode / RequiresDynamicCode` and is therefore unsuitable for Native AOT applications.

The new [IsReadOnly](#) property lets you check if the options instance is frozen.

Time abstraction

The new [TimeProvider](#) class and [ITimer](#) interface add *time abstraction* functionality, which allows you to mock time in test scenarios. In addition, you can use the time abstraction to mock [Task](#) operations that rely on time progression using [Task.Delay](#) and [Task.WaitAsync](#). The time abstraction supports the following essential time operations:

- Retrieve local and UTC time
- Obtain a timestamp for measuring performance
- Create a timer

The following code snippet shows some usage examples.

C#

```
// Get system time.
DateTimeOffset utcNow = TimeProvider.System.GetUtcNow();
DateTimeOffset localNow = TimeProvider.System.GetLocalNow();

// Create a time provider that works with a
// time zone that's different than the local time zone.
private class ZonedDateTimeProvider : TimeProvider
{
    private TimeZoneInfo _zoneInfo;

    public ZonedDateTimeProvider(TimeZoneInfo zoneInfo) : base()
    {
        _zoneInfo = zoneInfo ?? TimeZoneInfo.Local;
    }
}
```

```

    public override TimeZoneInfo LocalTimeZone => _zoneInfo;

    public static TimeProvider FromLocalTimeZone(TimeZoneInfo zoneInfo) =>
        new ZonedTimeProvider(zoneInfo);
}

// Create a timer using a time provider.
ITimer timer = timeProvider.CreateTimer(
    callBack, state, delay, Timeout.InfiniteTimeSpan);

// Measure a period using the system time provider.
long providerTimestamp1 = TimeProvider.System.GetTimestamp();
long providerTimestamp2 = TimeProvider.System.GetTimestamp();

var period = GetElapsedTime(providerTimestamp1, providerTimestamp2);

```

UTF8 improvements

If you want to enable writing out a string-like representation of your type to a destination span, implement the new [IUtf8SpanFormattable](#) interface on your type. This new interface is closely related to [ISpanFormattable](#), but targets UTF8 and `Span<byte>` instead of UTF16 and `Span<char>`.

[IUtf8SpanFormattable](#) has been implemented on all of the primitive types (plus others), with the exact same shared logic whether targeting `string`, `Span<char>`, or `Span<byte>`. It has full support for all formats (including the new "B" [binary specifier](#)) and all cultures. This means you can now format directly to UTF8 from `Byte`, `Complex`, `Char`, `DateOnly`, `DateTime`, `DateTimeOffset`, `Decimal`, `Double`, `Guid`, `Half`, `IPAddress`, `IPNetwork`, `Int16`, `Int32`, `Int64`, `Int128`, `IntPtr`, `NFloat`, `SByte`, `Single`, `Rune`, `TimeOnly`, `TimeSpan`, `UInt16`, `UInt32`, `UInt64`, `UInt128`, `UIntPtr`, and `Version`.

New [Utf8.TryWrite](#) methods provide a UTF8-based counterpart to the existing [MemoryExtensions.TryWrite](#) methods, which are UTF16-based. You can use interpolated string syntax to format a complex expression directly into a span of UTF8 bytes, for example:

C#

```

static bool FormatHexVersion(
    short major,
    short minor,
    short build,
    short revision,
    Span<byte> utf8Bytes,
    out int bytesWritten) =>
    Utf8.TryWrite(
        utf8Bytes,

```

```
CultureInfo.InvariantCulture,  
$"{major:X4}.{minor:X4}.{build:X4}.{revision:X4}",  
out bytesWritten);
```

The implementation recognizes [IUtf8SpanFormattable](#) on the format values and uses their implementations to write their UTF8 representations directly to the destination span.

The implementation also utilizes the new [Encoding.TryGetBytes\(ReadOnlySpan<Char>, Span<Byte>, Int32\)](#) method, which together with its [Encoding.TryGetChars\(ReadOnlySpan<Byte>, Span<Char>, Int32\)](#) counterpart, supports encoding and decoding into a destination span. If the span isn't long enough to hold the resulting state, the methods return `false` rather than throwing an exception.

Methods for working with randomness

The [System.Random](#) and [System.Security.Cryptography.RandomNumberGenerator](#) types introduce two new methods for working with randomness.

GetItems<T>()

The new [System.Random.GetItems](#) and [System.Security.Cryptography.RandomNumberGenerator.GetItems](#) methods let you randomly choose a specified number of items from an input set. The following example shows how to use `System.Random.GetItems<T>()` (on the instance provided by the [Random.Shared](#) property) to randomly insert 31 items into an array. This example could be used in a game of "Simon" where players must remember a sequence of colored buttons.

C#

```
private static ReadOnlySpan<Button> s_allButtons = new[]  
{  
    Button.Red,  
    Button.Green,  
    Button.Blue,  
    Button.Yellow,  
};  
  
// ...  
  
Button[] thisRound = Random.Shared.GetItems(s_allButtons, 31);  
// Rest of game goes here ...
```

Shuffle<T>()

The new [Random.Shuffle](#) and [RandomNumberGenerator.Shuffle<T>\(Span<T>\)](#) methods let you randomize the order of a span. These methods are useful for reducing training bias in machine learning (so the first thing isn't always training, and the last thing always test).

C#

```
YourType[] trainingData = LoadTrainingData();
Random.Shared.Shuffle(trainingData);

IDataView sourceData = mlContext.Data.LoadFromEnumerable(trainingData);

DataOperationsCatalog.TrainTestData split =
    mlContext.Data.TrainTestSplit(sourceData);
model = chain.Fit(split.TrainSet);

IDataView predictions = model.Transform(split.TestSet);
// ...
```

Performance-focused types

.NET 8 introduces several new types aimed at improving app performance.

- The new [System.Collections.Frozen](#) namespace includes the collection types [FrozenDictionary<TKey,TValue>](#) and [FrozenSet<T>](#). These types don't allow any changes to keys and values once a collection created. That requirement allows faster read operations (for example, `TryGetValue()`). These types are particularly useful for collections that are populated on first use and then persisted for the duration of a long-lived service, for example:

C#

```
private static readonly FrozenDictionary<string, bool>
s_configurationData =
    LoadConfigurationData().ToFrozenDictionary(optimizeForReads: true);

// ...
if (s_configurationData.TryGetValue(key, out bool setting) && setting)
{
    Process();
}
```

- The new [System.Buffers.SearchValues<T>](#) type is designed to be passed to methods that look for the first occurrence of any value in the passed collection. For

example, `String.IndexOfAny(Char[])` looks for the first occurrence of any character in the specified array in the `string` it's called on. .NET 8 adds new overloads of methods like `String.IndexOfAny` and `MemoryExtensions.IndexOfAny` that accept an instance of the new type. When you create an instance of `System.Buffers.SearchValues<T>`, all the data that's necessary to optimize subsequent searches is derived *at that time*, meaning the work is done up front.

- The new `System.Text.CompositeFormat` type is useful for optimizing format strings that aren't known at compile time (for example, if the format string is loaded from a resource file). A little extra time is spent up front to do work such as parsing the string, but it saves the work from being done on each use.

```
C#  
  
private static readonly CompositeFormat s_rangeMessage =  
    CompositeFormat.Parse(LoadRangeMessageResource());  
  
// ...  
static string GetMessage(int min, int max) =>  
    string.Format(CultureInfo.InvariantCulture, s_rangeMessage, min,  
    max);
```

- New `System.IO.Hashing.XxHash3` and `System.IO.Hashing.XxHash128` types provide implementations of the fast XXH3 and XXH128 hash algorithms.

System.Numerics and System.Runtime.Intrinsics

This section covers improvements to the `System.Numerics` and `System.Runtime.Intrinsics` namespaces.

- `Vector256<T>`, `Matrix3x2`, and `Matrix4x4` have improved hardware acceleration on .NET 8. For example, `Vector256<T>` was reimplemented to internally be `2x Vector128<T>` operations, where possible. This allows partial acceleration of some functions when `Vector128.IsHardwareAccelerated == true` but `Vector256.IsHardwareAccelerated == false`, such as on Arm64.
- Hardware intrinsics are now annotated with the `ConstExpected` attribute. This ensures that users are aware when the underlying hardware expects a constant and therefore when a non-constant value may unexpectedly hurt performance.
- The `Lerp(TSelf, TSelf, TSelf)` `Lerp` API has been added to `IFloatingPointeee754<TSelf>` and therefore to `float` (`Single`), `double` (`Double`), and `Half`. This API allows a linear interpolation between two values to be performed efficiently and correctly.

Vector512 and AVX-512

.NET Core 3.0 expanded SIMD support to include the platform-specific hardware intrinsics APIs for x86/x64. .NET 5 added support for Arm64 and .NET 7 added the cross-platform hardware intrinsics. .NET 8 furthers SIMD support by introducing [Vector512<T>](#) and support for [Intel Advanced Vector Extensions 512 \(AVX-512\)](#) instructions.

Specifically, .NET 8 includes support for the following key features of AVX-512:

- 512-bit vector operations
- Additional 16 SIMD registers
- Additional instructions available for 128-bit, 256-bit, and 512-bit vectors

If you have hardware that supports the functionality, then

[Vector512.IsHardwareAccelerated](#) now reports `true`.

.NET 8 also adds several platform-specific classes under the

[System.Runtime.Intrinsics.X86](#) namespace:

- [Avx512F](#) (foundational)
- [Avx512BW](#) (byte and word)
- [Avx512CD](#) (conflict detection)
- [Avx512DQ](#) (doubleword and quadword)
- [Avx512Vbmi](#) (vector byte manipulation instructions)

These classes follow the same general shape as other instruction set architectures (ISAs) in that they expose an [IsSupported](#) property and a nested [Avx512F.X64](#) class for instructions available only to 64-bit processes. Additionally, each class has a nested [Avx512F.VL](#) class that exposes the [Avx512VL](#) (vector length) extensions for the corresponding instruction set.

Even if you don't explicitly use `Vector512`-specific or `Avx512F`-specific instructions in your code, you'll likely still benefit from the new AVX-512 support. The JIT can take advantage of the additional registers and instructions implicitly when using [Vector128<T>](#) or [Vector256<T>](#). The base class library uses these hardware intrinsics internally in most operations exposed by [Span<T>](#) and [ReadOnlySpan<T>](#) and in many of the math APIs exposed for the primitive types.

Data validation

The [System.ComponentModel.DataAnnotations](#) namespace includes new data validation attributes intended for validation scenarios in cloud-native services. While the pre-existing `DataAnnotations` validators are geared towards typical UI data-entry validation,

such as fields on a form, the new attributes are designed to validate non-user-entry data, such as [configuration options](#). In addition to the new attributes, new properties were added to the [RangeAttribute](#) and [RequiredAttribute](#) types.

[+] [Expand table](#)

| New API | Description |
|---|---|
| RangeAttribute.MinimumIsExclusive RangeAttribute.MaximumIsExclusive | Specifies whether bounds are included in the allowable range. |
| System.ComponentModel.DataAnnotations.LengthAttribute | Specifies both lower and upper bounds for strings or collections. For example, <code>[Length(10, 20)]</code> requires at least 10 elements and at most 20 elements in a collection. |
| System.ComponentModel.DataAnnotations.Base64StringAttribute | Validates that a string is a valid Base64 representation. |
| System.ComponentModel.DataAnnotations.AllowedValuesAttribute System.ComponentModel.DataAnnotations.DeniedValuesAttribute | Specify allow lists and deny lists, respectively. For example, <code>[AllowedValues("apple", "banana", "mango")]</code> . |

Metrics

New APIs let you attach key-value pair tags to [Meter](#) and [Instrument](#) objects when you create them. Aggregators of published metric measurements can use the tags to differentiate the aggregated values.

C#

```
var options = new MeterOptions("name")
{
    Version = "version",
    // Attach these tags to the created meter
    Tags = new TagList()
    {
        { "MeterKey1", "MeterValue1" },
        { "MeterKey2", "MeterValue2" }
    }
};
```

```
Meter meter = meterFactory.Create(options);

Instrument instrument = meter.CreateCounter<int>(
    "counter", null, null, new TagList() { { "counterKey1", "counterValue1" } });
};

instrument.Add(1);
```

The new APIs include:

- `MeterOptions`
- `Meter(MeterOptions)`
- `CreateCounter<T>(String, String, String, IEnumerable<KeyValuePair<String, Object>>)`

Cryptography

.NET 8 adds support for the SHA-3 hashing primitives. (SHA-3 is currently supported by Linux with OpenSSL 1.1.1 or later and Windows 11 Build 25324 or later.) APIs where SHA-2 is available now offer a SHA-3 compliment. This includes `SHA3_256`, `SHA3_384`, and `SHA3_512` for hashing; `HMACSHA3_256`, `HMACSHA3_384`, and `HMACSHA3_512` for HMAC; `HashAlgorithmName.SHA3_256`, `HashAlgorithmName.SHA3_384`, and `HashAlgorithmName.SHA3_512` for hashing where the algorithm is configurable; and `RSAEncryptionPadding.OaepSHA3_256`, `RSAEncryptionPadding.OaepSHA3_384`, and `RSAEncryptionPadding.OaepSHA3_512` for RSA OAEP encryption.

The following example shows how to use the APIs, including the `SHA3_256.IsEnabled` property to determine if the platform supports SHA-3.

C#

```
// Hashing example
if (SHA3_256.IsEnabled)
{
    byte[] hash = SHA3_256.HashData(dataToHash);
}
else
{
    // ...
}

// Signing example
if (SHA3_256.IsEnabled)
{
    using ECDsa ec = ECDsa.Create(ECCurve.NamedCurves.nistP256);
    byte[] signature = ec.SignData(dataToBeSigned,
        HashAlgorithmName.SHA3_256);
```

```
    }
    else
    {
        // ...
    }
```

SHA-3 support is currently aimed at supporting cryptographic primitives. Higher-level constructions and protocols aren't expected to fully support SHA-3 initially. These protocols include X.509 certificates, [SignedXml](#), and COSE.

Networking

Support for HTTPS proxy

Until now, the proxy types that [HttpClient](#) supported all allowed a "man-in-the-middle" to see which site the client is connecting to, even for HTTPS URIs. [HttpClient](#) now supports *HTTPS proxy*, which creates an encrypted channel between the client and the proxy so all requests can be handled with full privacy.

To enable HTTPS proxy, set the `all_proxy` environment variable, or use the [WebProxy](#) class to control the proxy programmatically.

```
Unix: export all_proxy=https://x.x.x.x:3218 Windows: set  
all_proxy=https://x.x.x.x:3218
```

You can also use the [WebProxy](#) class to control the proxy programmatically.

Stream-based ZipFile methods

.NET 8 includes new overloads of [ZipFile.CreateDirectory](#) that allow you to collect all the files included in a directory and zip them, then store the resulting zip file into the provided stream. Similarly, new [ZipFile.ExtractToDirectory](#) overloads let you provide a stream containing a zipped file and extract its contents into the filesystem. These are the new overloads:

C#

```
namespace System.IO.Compression;  
  
public static partial class ZipFile  
{  
    public static void CreateFromDirectory(  
        string sourceDirectoryName, Stream destination);
```

```

    public static void CreateFromDirectory(
        string sourceDirectoryName,
        Stream destination,
        CompressionLevel compressionLevel,
        bool includeBaseDirectory);

    public static void CreateFromDirectory(
        string sourceDirectoryName,
        Stream destination,
        CompressionLevel compressionLevel,
        bool includeBaseDirectory,
        Encoding? entryNameEncoding);

    public static void ExtractToDirectory(
        Stream source, string destinationDirectoryName) { }

    public static void ExtractToDirectory(
        Stream source, string destinationDirectoryName, bool overwriteFiles)
{ }

    public static void ExtractToDirectory(
        Stream source, string destinationDirectoryName, Encoding?
entryNameEncoding) { }

    public static void ExtractToDirectory(
        Stream source, string destinationDirectoryName, Encoding?
entryNameEncoding, bool overwriteFiles) { }
}

```

These new APIs can be useful when disk space is constrained, because they avoid having to use the disk as an intermediate step.

Extension libraries

This section contains the following subtopics:

- Options validation
- LoggerMessageAttribute constructors
- Extensions metrics
- Hosted lifecycle services
- Keyed DI services
- System.Numerics.Tensors.TensorPrimitives

Keyed DI services

Keyed dependency injection (DI) services provide a means for registering and retrieving DI services using keys. By using keys, you can scope how you register and consume

services. These are some of the new APIs:

- The [IKeyedServiceProvider](#) interface.
- The [ServiceKeyAttribute](#) attribute, which can be used to inject the key that was used for registration/resolution in the constructor.
- The [FromKeyedServicesAttribute](#) attribute, which can be used on service constructor parameters to specify which keyed service to use.
- Various new extension methods for [IServiceCollection](#) to support keyed services, for example, [ServiceCollectionServiceExtensions.AddKeyedScoped](#).
- The [ServiceProvider](#) implementation of [IKeyedServiceProvider](#).

The following example shows you how to use keyed DI services.

C#

```
var builder = WebApplication.CreateBuilder(args);
builder.Services.AddSingleton<BigCacheConsumer>();
builder.Services.AddSingleton<SmallCacheConsumer>();
builder.Services.AddKeyedSingleton<ICache, BigCache>("big");
builder.Services.AddKeyedSingleton<ICache, SmallCache>("small");
var app = builder.Build();
app.MapGet("/big", (BigCacheConsumer data) => data.GetData());
app.MapGet("/small", (SmallCacheConsumer data) => data.GetData());
app.MapGet("/big-cache", ([FromKeyedServices("big")] ICache cache) =>
cache.Get("data"));
app.MapGet("/small-cache", (HttpContext httpContext) =>
httpContext.RequestServices.GetRequiredKeyedService<ICache>(
("small").Get("data"));
app.Run();

class BigCacheConsumer([FromKeyedServices("big")] ICache cache)
{
    public object? GetData() => cache.Get("data");
}

class SmallCacheConsumer(IServiceProvider serviceProvider)
{
    public object? GetData() =>
serviceProvider.GetRequiredKeyedService<ICache>("small").Get("data");
}

public interface ICache
{
    object Get(string key);
}

public class BigCache : ICache
{
    public object Get(string key) => $"Resolving {key} from big cache.";
}
```

```
public class SmallCache : ICache
{
    public object Get(string key) => $"Resolving {key} from small cache.";
}
```

For more information, see [dotnet/runtime#64427](#).

Hosted lifecycle services

Hosted services now have more options for execution during the application lifecycle. [IHostedService](#) provided `StartAsync` and `StopAsync`, and now [IHostedLifecycleService](#) provides these additional methods:

- `StartingAsync(CancellationToken)`
- `StartedAsync(CancellationToken)`
- `StoppingAsync(CancellationToken)`
- `StoppedAsync(CancellationToken)`

These methods run before and after the existing points respectively.

The following example shows how to use the new APIs.

C#

```
using Microsoft.Extensions.Hosting;

IHostBuilder hostBuilder = new HostBuilder();
hostBuilder.ConfigureServices(services =>
{
    services.AddHostedService<MyService>();
});

using (IHost host = hostBuilder.Build())
{
    await host.StartAsync();
}

public class MyService : IHostedLifecycleService
{
    public Task StartingAsync(CancellationToken cancellationToken) => /* add
logic here */ Task.CompletedTask;
    public Task StartAsync(CancellationToken cancellationToken) => /* add
logic here */ Task.CompletedTask;
    public Task StartedAsync(CancellationToken cancellationToken) => /* add
logic here */ Task.CompletedTask;
    public Task StopAsync(CancellationToken cancellationToken) => /* add
logic here */ Task.CompletedTask;
    public Task StoppedAsync(CancellationToken cancellationToken) => /* add
logic here */ Task.CompletedTask;
}
```

```
    public Task StoppingAsync(CancellationToken cancellationToken) => /* add
logic here */ Task.CompletedTask;
}
```

For more information, see [dotnet/runtime#86511](#).

Options validation

Source generator

To reduce startup overhead and improve validation feature set, we've introduced a source code generator that implements the validation logic. The following code shows example models and validator classes.

C#

```
public class FirstModelNoNamespace
{
    [Required]
    [MinLength(5)]
    public string P1 { get; set; } = string.Empty;

    [Microsoft.Extensions.Options.ValidateObjectMembers(
        typeof(SecondValidatorNoNamespace))]
    public SecondModelNoNamespace? P2 { get; set; }
}

public class SecondModelNoNamespace
{
    [Required]
    [MinLength(5)]
    public string P4 { get; set; } = string.Empty;
}

[OptionsValidator]
public partial class FirstValidatorNoNamespace
    : IValidateOptions<FirstModelNoNamespace>
{

}

[OptionsValidator]
public partial class SecondValidatorNoNamespace
    : IValidateOptions<SecondModelNoNamespace>
{}
```

If your app uses dependency injection, you can inject the validation as shown in the following example code.

C#

```
var builder = WebApplication.CreateBuilder(args);
builder.Services.AddControllersWithViews();
builder.Services.Configure<FirstModelNoNamespace>(
    builder.Configuration.GetSection(...));

builder.Services.AddSingleton<
    IValidateOptions<FirstModelNoNamespace>, FirstValidatorNoNamespace>();
builder.Services.AddSingleton<
    IValidateOptions<SecondModelNoNamespace>, SecondValidatorNoNamespace>();
```

ValidateOptionsResultBuilder type

.NET 8 introduces the [ValidateOptionsResultBuilder](#) type to facilitate the creation of a [ValidateOptionsResult](#) object. Importantly, this builder allows for the accumulation of multiple errors. Previously, creating the [ValidateOptionsResult](#) object that's required to implement `IValidateOptions<TOptions>.Validate(String, TOptions)` was difficult and sometimes resulted in layered validation errors. If there were multiple errors, the validation process often stopped at the first error.

The following code snippet shows an example usage of [ValidateOptionsResultBuilder](#).

C#

```
ValidateOptionsResultBuilder builder = new();
builder.AddError("Error: invalid operation code");
builder.AddResult(ValidateOptionsResult.Fail("Invalid request parameters"));
builder.AddError("Malformed link", "Url");

// Build ValidateOptionsResult object has accumulating multiple errors.
ValidateOptionsResult result = builder.Build();

// Reset the builder to allow using it in new validation operation.
builder.Clear();
```

LoggerMessageAttribute constructors

[LoggerMessageAttribute](#) now offers additional constructor overloads. Previously, you had to choose either the parameterless constructor or the constructor that required all of the parameters (event ID, log level, and message). The new overloads offer greater flexibility in specifying the required parameters with reduced code. If you don't supply an event ID, the system generates one automatically.

C#

```
public LoggerMessageAttribute(LogLevel level, string message);
public LoggerMessageAttribute(LogLevel level);
public LoggerMessageAttribute(string message);
```

Extensions metrics

IMeterFactory interface

You can register the new [IMeterFactory](#) interface in dependency injection (DI) containers and use it to create [Meter](#) objects in an isolated manner.

Register the [IMeterFactory](#) to the DI container using the default meter factory implementation:

C#

```
// 'services' is the DI IServiceCollection.
services.AddMetrics();
```

Consumers can then obtain the meter factory and use it to create a new [Meter](#) object.

C#

```
IMeterFactory meterFactory =
serviceProvider.GetRequiredService<IMeterFactory>();

MeterOptions options = new MeterOptions("MeterName")
{
    Version = "version",
};

Meter meter = meterFactory.Create(options);
```

MetricCollector<T> class

The new [MetricCollector<T>](#) class lets you record metric measurements along with timestamps. Additionally, the class offers the flexibility to use a time provider of your choice for accurate timestamp generation.

C#

```
const string CounterName = "MyCounter";

var now = DateTimeOffset.Now;
```

```

var timeProvider = new FakeTimeProvider(now);
using var meter = new Meter(Guid.NewGuid().ToString());
var counter = meter.CreateCounter<long>(CounterName);
using var collector = new MetricCollector<long>(counter, timeProvider);

Assert.Empty(collector.GetMeasurementSnapshot());
Assert.Null(collector.LastMeasurement);

counter.Add(3);

// Verify the update was recorded.
Assert.Equal(counter, collector.Instrument);
Assert.NotNull(collector.LastMeasurement);

Assert.Single(collector.GetMeasurementSnapshot());
Assert.Same(collector.GetMeasurementSnapshot().Last(),
collector.LastMeasurement);
Assert.Equal(3, collector.LastMeasurement.Value);
Assert.Empty(collector.LastMeasurement.Tags);
Assert.Equal(now, collector.LastMeasurement.Timestamp);

```

System.Numerics.Tensors.TensorPrimitives

The updated [System.Numerics.Tensors](#) NuGet package includes APIs in the new [TensorPrimitives](#) namespace that add support for tensor operations. The tensor primitives optimize data-intensive workloads like those of AI and machine learning.

AI workloads like semantic search and retrieval-augmented generation (RAG) extend the natural language capabilities of large language models such as ChatGPT by augmenting prompts with relevant data. For these workloads, operations on vectors—like *cosine similarity* to find the most relevant data to answer a question—are crucial. The System.Numerics.Tensors.TensorPrimitives package provides APIs for vector operations, meaning you don't need to take an external dependency or write your own implementation.

This package replaces the [System.Numerics.Tensors package](#).

For more information, see the [Announcing .NET 8 RC 2 blog post](#).

Garbage collection

.NET 8 adds a capability to adjust the memory limit on the fly. This is useful in cloud-service scenarios, where demand comes and goes. To be cost-effective, services should scale up and down on resource consumption as the demand fluctuates. When a service detects a decrease in demand, it can scale down resource consumption by reducing its

memory limit. Previously, this would fail because the garbage collector (GC) was unaware of the change and might allocate more memory than the new limit. With this change, you can call the [RefreshMemoryLimit\(\)](#) API to update the GC with the new memory limit.

There are some limitations to be aware of:

- On 32-bit platforms (for example, Windows x86 and Linux ARM), .NET is unable to establish a new heap hard limit if there isn't already one.
- The API might return a non-zero status code indicating the refresh failed. This can happen if the scale-down is too aggressive and leaves no room for the GC to maneuver. In this case, consider calling `GC.Collect(2, GCCollectionMode.Aggressive)` to shrink the current memory usage, and then try again.
- If you scale up the memory limit beyond the size that the GC believes the process can handle during startup, the `RefreshMemoryLimit` call will succeed, but it won't be able to use more memory than what it perceives as the limit.

The following code snippet shows how to call the API.

```
C#
```

```
GC.RefreshMemoryLimit();
```

You can also refresh some of the GC configuration settings related to the memory limit. The following code snippet sets the heap hard limit to 100 mebibytes (MiB):

```
C#
```

```
AppContext.SetData("GCHardLimit", (ulong)100 * 1_024 * 1_024);
GC.RefreshMemoryLimit();
```

The API can throw an [InvalidOperationException](#) if the hard limit is invalid, for example, in the case of negative heap hard limit percentages and if the hard limit is too low. This can happen if the heap hard limit that the refresh will set, either because of new AppData settings or implied by the container memory limit changes, is lower than what's already committed.

Configuration-binding source generator

.NET 8 introduces a source generator to provide AOT and trim-friendly [configuration](#) in ASP.NET Core. The generator is an alternative to the pre-existing reflection-based

implementation.

The source generator probes for `Configure(TOptions)`, `Bind`, and `Get` calls to retrieve type info from. When the generator is enabled in a project, the compiler implicitly chooses generated methods over the pre-existing reflection-based framework implementations.

No source code changes are needed to use the generator. It's enabled by default in AOT'd web apps. For other project types, the source generator is off by default, but you can opt in by setting the `EnableConfigurationBindingGenerator` property to `true` in your project file:

XML

```
<PropertyGroup>
  <EnableConfigurationBindingGenerator>true</EnableConfigurationBindingGenerator>
  or
</PropertyGroup>
```

The following code shows an example of invoking the binder.

C#

```
using Microsoft.AspNetCore.Builder;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;

WebApplicationBuilder builder = WebApplication.CreateBuilder(args);
IConfigurationSection section =
builder.Configuration.GetSection("MyOptions");

// !! Configure call - to be replaced with source-gen'd implementation
builder.Services.Configure<MyOptions>(section);

// !! Get call - to be replaced with source-gen'd implementation
MyOptions options0 = section.Get<MyOptions>();

// !! Bind call - to be replaced with source-gen'd implementation
MyOptions options1 = new MyOptions();
section.Bind(options1);

WebApplication app = builder.Build();
app.MapGet("/", () => "Hello World!");
app.Run();

public class MyOptions
{
    public int A { get; set; }
    public string S { get; set; }
```

```
    public byte[] Data { get; set; }
    public Dictionary<string, string> Values { get; set; }
    public List<MyClass> Values2 { get; set; }
}

public class MyClass
{
    public int SomethingElse { get; set; }
}
```

Reflection improvements

Function pointers were introduced in .NET 5, however, the corresponding support for reflection wasn't added at that time. When using `typeof` or reflection on a function pointer, for example, `typeof(delegate*<void>())` or `FieldInfo.FieldType` respectively, an `IntPtr` was returned. Starting in .NET 8, a `System.Type` object is returned instead. This type provides access to function pointer metadata, including the calling conventions, return type, and parameters.

ⓘ Note

A function pointer instance, which is a physical address to a function, continues to be represented as an `IntPtr`. Only the reflection type has changed.

The new functionality is currently implemented only in the CoreCLR runtime and `MetadataLoadContext`.

New APIs have been added to `System.Type`, such as `IsFunctionPointer`, and to `System.Reflection.PropertyInfo`, `System.Reflection.FieldInfo`, and `System.Reflection.ParameterInfo`. The following code shows how to use some of the new APIs for reflection.

C#

```
// Sample class that contains a function pointer field.
public unsafe class UClass
{
    public delegate* unmanaged[Cdecl, SuppressGCTransition]<in int, void>
_fp;
}

// ...

FieldInfo fieldInfo = typeof(UClass).GetField(nameof(UClass._fp));
```

```

// Obtain the function pointer type from a field.
Type fpType = fieldInfo.FieldType;

// New methods to determine if a type is a function pointer.
Console.WriteLine(
    $"IsFunctionPointer: {fpType.IsFunctionPointer}");
Console.WriteLine(
    $"IsUnmanagedFunctionPointer: {fpType.IsUnmanagedFunctionPointer}");

// New methods to obtain the return and parameter types.
Console.WriteLine($"Return type: {fpType.GetFunctionPointerReturnType()}");

foreach (Type parameterType in fpType.GetFunctionPointerParameterTypes())
{
    Console.WriteLine($"Parameter type: {parameterType}");
}

// Access to custom modifiers and calling conventions requires a "modified
// type".
Type modifiedType = fieldInfo.GetModifiedFieldType();

// A modified type forwards most members to its underlying type.
Type normalType = modifiedType.UnderlyingSystemType;

// New method to obtain the calling conventions.
foreach (Type callConv in
modifiedType.GetFunctionPointerCallingConventions())
{
    Console.WriteLine($"Calling convention: {callConv}");
}

// New method to obtain the custom modifiers.
var modifiers =
    modifiedType.GetFunctionPointerParameterTypes()
[0].GetRequiredCustomModifiers();

foreach (Type modreq in modifiers)
{
    Console.WriteLine($"Required modifier for first parameter: {modreq}");
}

```

The previous example produces the following output:

Output

```

IsFunctionPointer: True
IsUnmanagedFunctionPointer: True
Return type: System.Void
Parameter type: System.Int32&
Calling convention:
System.Runtime.CompilerServices.CallConvSuppressGCTransition
Calling convention: System.Runtime.CompilerServices.CallConvCdecl

```

Required modifier for first parameter:
System.Runtime.InteropServices.InAttribute

Native AOT support

The option to [publish as Native AOT](#) was first introduced in .NET 7. Publishing an app with Native AOT creates a fully self-contained version of your app that doesn't need a runtime—everything is included in a single file. .NET 8 brings the following improvements to Native AOT publishing:

- Adds support for the x64 and Arm64 architectures on *macOS*.
- Reduces the sizes of Native AOT apps on Linux by up to 50%. The following table shows the size of a "Hello World" app published with Native AOT that includes the entire .NET runtime on .NET 7 vs. .NET 8:

[+] [Expand table](#)

| Operating system | .NET 7 | .NET 8 |
|---|---------|---------|
| Linux x64 (with <code>-p:StripSymbols=true</code>) | 3.76 MB | 1.84 MB |
| Windows x64 | 2.85 MB | 1.77 MB |

- Lets you specify an optimization preference: size or speed. By default, the compiler chooses to generate fast code while being mindful of the size of the application. However, you can use the `<OptimizationPreference>` MSBuild property to optimize specifically for one or the other. For more information, see [Optimize AOT deployments](#).

Console app template

The default console app template now includes support for AOT out-of-the-box. To create a project that's configured for AOT compilation, just run `dotnet new console --aot`. The project configuration added by `--aot` has three effects:

- Generates a native self-contained executable with Native AOT when you publish the project, for example, with `dotnet publish` or Visual Studio.
- Enables compatibility analyzers for trimming, AOT, and single file. These analyzers alert you to potentially problematic parts of your project (if there are any).
- Enables debug-time emulation of AOT so that when you debug your project without AOT compilation, you get a similar experience to AOT. For example, if you

use `System.Reflection.Emit` in a NuGet package that wasn't annotated for AOT (and therefore was missed by the compatibility analyzer), the emulation means you won't have any surprises when you try to publish the project with AOT.

Target iOS-like platforms with Native AOT

.NET 8 starts the work to enable Native AOT support for iOS-like platforms. You can now build and run .NET iOS and .NET MAUI applications with Native AOT on the following platforms:

- `ios`
- `iossimulator`
- `maccatalyst`
- `tvos`
- `tvossimulator`

Preliminary testing shows that app size on disk decreases by about 35% for .NET iOS apps that use Native AOT instead of Mono. App size on disk for .NET MAUI iOS apps decreases up to 50%. Additionally, the startup time is also faster. .NET iOS apps have about 28% faster startup time, while .NET MAUI iOS apps have about 50% better startup performance compared to Mono. The .NET 8 support is experimental and only the first step for the feature as a whole. For more information, see the [.NET 8 Performance Improvements in .NET MAUI blog post ↗](#).

Native AOT support is available as an opt-in feature intended for app deployment; Mono is still the default runtime for app development and deployment. To build and run a .NET MAUI application with Native AOT on an iOS device, use `dotnet workload install maui` to install the .NET MAUI workload and `dotnet new maui -n HelloMaui` to create the app. Then, set the MSBuild property `PublishAot` to `true` in the project file.

XML

```
<PropertyGroup>
  <PublishAot>true</PublishAot>
</PropertyGroup>
```

When you set the required property and run `dotnet publish` as shown in the following example, the app will be deployed by using Native AOT.

.NET CLI

```
dotnet publish -f net8.0-ios -c Release -r ios-arm64 /t:Run
```

Limitations

Not all iOS features are compatible with Native AOT. Similarly, not all libraries commonly used in iOS are compatible with NativeAOT. And in addition to the existing [limitations of Native AOT deployment](#), the following list shows some of the other limitations when targeting iOS-like platforms:

- Using Native AOT is only enabled during app deployment (`dotnet publish`).
- Managed code debugging is only supported with Mono.
- Compatibility with the .NET MAUI framework is limited.

Performance improvements

.NET 8 includes improvements to code generation and just-in time (JIT) compilation:

- Arm64 performance improvements
- SIMD improvements
- Support for AVX-512 ISA extensions (see [Vector512 and AVX-512](#))
- Cloud-native improvements
- JIT throughput improvements
- Loop and general optimizations
- Optimized access for fields marked with [ThreadStaticAttribute](#)
- Consecutive register allocation. Arm64 has two instructions for table vector lookup, which require that all entities in their tuple operands are present in consecutive registers.
- JIT/NativeAOT can now unroll and auto-vectorize some memory operations with SIMD, such as comparison, copying, and zeroing, if it can determine their sizes at compile time.

In addition, dynamic profile-guided optimization (PGO) has been improved and is now enabled by default. You no longer need to use a [runtime configuration option](#) to enable it. Dynamic PGO works hand-in-hand with tiered compilation to further optimize code based on additional instrumentation that's put in place during tier 0.

On average, dynamic PGO increases performance by about 15%. In a benchmark suite of ~4600 tests, 23% saw performance improvements of 20% or more.

Codegen struct promotion

.NET 8 includes a new physical promotion optimization pass for codegen that generalizes the JIT's ability to promote struct variables. This optimization (also called

(scalar replacement of aggregates) replaces the fields of struct variables by primitive variables that the JIT is then able to reason about and optimize more precisely.

The JIT already supported this optimization but with several large limitations including:

- It was only supported for structs with four or fewer fields.
- It was only supported if each field was a primitive type, or a simple struct wrapping a primitive type.

Physical promotion removes these limitations, which fixes a number of long-standing JIT issues.

.NET MAUI

For information about what's new in .NET MAUI in .NET 8, see [What's new in .NET MAUI for .NET 8](#).

.NET SDK

This section contains the following subtopics:

- [CLI-based project evaluation](#)
- [Terminal build output](#)
- [Simplified output paths](#)
- ['dotnet workload clean' command](#)
- ['dotnet publish' and 'dotnet pack' assets](#)
- [Template engine](#)
- [Source Link](#)
- [Source-build SDK](#)

CLI-based project evaluation

MSBuild includes a new feature that makes it easier to incorporate data from MSBuild into your scripts or tools. The following new flags are available for CLI commands such as [dotnet publish](#) to obtain data for use in CI pipelines and elsewhere.

 [Expand table](#)

| Flag | Description |
|------------------------------|---|
| --getProperty:<PROPERTYNAME> | Retrieves the MSBuild property with the specified name. |
| --getItem:<ITEMTYPE> | Retrieves MSBuild items of the specified type. |

| Flag | Description |
|---------------------------------|--|
| --getTargetResults:<TARGETNAME> | Retrieves the outputs from running the specified target. |

Values are written to the standard output. Multiple or complex values are output as JSON, as shown in the following examples.

.NET CLI

```
>dotnet publish --getProperty:OutputPath
bin\Release\net8.0\
```

.NET CLI

```
>dotnet publish -p PublishProfile=DefaultContainer --
getProperty:GeneratedContainerDigest --
getProperty:GeneratedContainerConfiguration
{
  "Properties": {
    "GeneratedContainerDigest":
    "sha256:ef880a503bbabcb84bbb6a1aa9b41b36dc1ba08352e7cd91c0993646675174c4",
    "GeneratedContainerConfiguration": "{\u0022config\u0022:
{\u0022ExposedPorts\u0022:{\u00228080/tcp\u0022:{}}, \u0022Labels\u0022...}}"
  }
}
```

.NET CLI

```
>dotnet publish -p PublishProfile=DefaultContainer --
getItem:ContainerImageTags
{
  "Items": {
    "ContainerImageTags": [
      {
        "Identity": "latest",
        ...
      ]
    }
  }
}
```

Terminal build output

`dotnet build` has a new option to produce more modernized build output. This *terminal logger* output groups errors with the project they came from, better differentiates the different target frameworks for multi-targeted projects, and provides real-time information about what the build is doing. To opt into the new output, use the `--t1` option. For more information about this option, see [dotnet build options](#).

Simplified output paths

.NET 8 introduces an option to simplify the output path and folder structure for build outputs. Previously, .NET apps produced a deep and complex set of output paths for different build artifacts. The new, simplified output path structure gathers all build outputs into a common location, which makes it easier for tooling to anticipate.

For more information, see [Artifacts output layout](#).

dotnet workload clean command

.NET 8 introduces a new command to clean up workload packs that might be left over through several .NET SDK or Visual Studio updates. If you encounter issues when managing workloads, consider using `workload clean` to safely restore to a known state before trying again. The command has two modes:

- `dotnet workload clean`

Runs [workload garbage collection](#) for file-based or MSI-based workloads, which cleans up orphaned packs. Orphaned packs are from uninstalled versions of the .NET SDK or packs where installation records for the pack no longer exist.

If Visual Studio is installed, the command also lists any workloads that you should clean up manually using Visual Studio.

- `dotnet workload clean --all`

This mode is more aggressive and cleans every pack on the machine that's of the current SDK workload installation type (and that's not from Visual Studio). It also removes all workload installation records for the running .NET SDK feature band and below.

dotnet publish and dotnet pack assets

Since the `dotnet publish` and `dotnet pack` commands are intended to produce production assets, they now produce `Release` assets by default.

The following output shows the different behavior between `dotnet build` and `dotnet publish`, and how you can revert to publishing `Debug` assets by setting the `PublishRelease` property to `false`.

Console

```
/app# dotnet new console  
/app# dotnet build  
  app -> /app/bin/Debug/net8.0/app.dll  
/app# dotnet publish  
  app -> /app/bin/Release/net8.0/app.dll  
  app -> /app/bin/Release/net8.0/publish/  
/app# dotnet publish -p:PublishRelease=false  
  app -> /app/bin/Debug/net8.0/app.dll  
  app -> /app/bin/Debug/net8.0/publish/
```

For more information, see '[dotnet pack](#)' uses Release config and '[dotnet publish](#)' uses Release config.

dotnet restore security auditing

Starting in .NET 8, you can opt into security checks for known vulnerabilities when dependency packages are restored. This auditing produces a report of security vulnerabilities with the affected package name, the severity of the vulnerability, and a link to the advisory for more details. When you run `dotnet add` or `dotnet restore`, warnings NU1901-NU1904 will appear for any vulnerabilities that are found. For more information, see [Audit for security vulnerabilities](#).

Template engine

The [template engine](#) provides a more secure experience in .NET 8 by integrating some of NuGet's security-related features. The improvements include:

- Prevent downloading packages from `http://` feeds by default. For example, the following command will fail to install the template package because the source URL doesn't use HTTPS.

```
dotnet new install console --add-source  
"http://pkgs.dev.azure.com/dnceng/public/_packaging/dotnet-  
public/nuget/v3/index.json"
```

You can override this limitation by using the `--force` flag.

- For `dotnet new`, `dotnet new install`, and `dotnet new update`, check for known vulnerabilities in the template package. If vulnerabilities are found and you wish to proceed, you must use the `--force` flag.
- For `dotnet new`, provide information about the template package owner. Ownership is verified by the NuGet portal and can be considered a trustworthy

characteristic.

- For `dotnet search` and `dotnet uninstall`, indicate whether a template is installed from a package that's "trusted"—that is, it uses a [reserved prefix](#).

Source Link

[Source Link](#) is now included in the .NET SDK. The goal is that by bundling Source Link into the SDK, instead of requiring a separate `<PackageReference>` for the package, more packages will include this information by default. That information will improve the IDE experience for developers.

Note

As a side effect of this change, commit information is included in the `InformationalVersion` value of built libraries and applications, even those that target .NET 7 or an earlier version. For more information, see [Source Link included in the .NET SDK](#).

Source-build SDK

The Linux distribution-built (source-build) SDK now has the capability to build self-contained applications using the source-build runtime packages. The distribution-specific runtime package is bundled with the source-build SDK. During self-contained deployment, this bundled runtime package will be referenced, thereby enabling the feature for users.

Globalization

HybridGlobalization mode on iOS/tvOS/MacCatalyst

Mobile apps can now use a new *hybrid* globalization mode that uses a lighter ICU bundle. In hybrid mode, globalization data is partially pulled from the ICU bundle and partially from calls into Native API. It serves all the [locales supported by mobile](#).

`HybridGlobalization` is most suitable for apps that can't work in `InvariantGlobalization` mode and that use cultures that were trimmed from ICU data on mobile. You can also use it when you want to load a smaller ICU data file. (The `icudt_hybrid.dat` file is 34.5 % smaller than the default ICU data file `icudt.dat`.)

To use `HybridGlobalization` mode, set the MSBuild property to true:

XML

```
<PropertyGroup>
  <HybridGlobalization>true</HybridGlobalization>
</PropertyGroup>
```

There are some limitations to be aware of:

- Due to limitations of Native API, not all globalization APIs are supported in hybrid mode.
- Some of the supported APIs have different behavior.

To make sure your application isn't affected, see [Behavioral differences ↗](#).

Containers

- [Container images](#)
- [Container publishing](#)

Container images

The following changes have been made to .NET container images for .NET 8:

- [Generated-image defaults](#)
- [Debian 12](#)
- [Non-root user](#)
- [Chiseled Ubuntu images](#)
- [Build multi-platform container images](#)
- [ASP.NET composite images](#)

Generated-image defaults

The new [non-root capability](#) of the Microsoft .NET containers is now the default, which helps your apps stay secure-by-default. Change this default at any time by setting your own `ContainerUser`.

The default container tag is now `latest`. This default is in line with other tooling in the containers space and makes containers easier to use in inner development loops.

Debian 12

The container images now use [Debian 12 \(Bookworm\)](#). Debian is the default Linux distro in the .NET container images.

Non-root user

Images include a `non-root` user. This user makes the images `non-root` capable. To run as `non-root`, add the following line at the end of your Dockerfile (or a similar instruction in your Kubernetes manifests):

```
Dockerfile
USER app
```

.NET 8 adds an environment variable for the UID for the `non-root` user, which is 64198. This environment variable is useful for the Kubernetes `runAsNonRoot` test, which requires that the container user be set via UID and not by name. This [dockerfile](#) shows an example usage.

The default port also changed from port `80` to `8080`. To support this change, a new environment variable `ASPNETCORE_HTTP_PORTS` is available to make it easier to change ports. The variable accepts a list of ports, which is simpler than the format required by `ASPNETCORE_URLS`. If you change the port back to port `80` using one of these variables, you can't run as `non-root`.

Chiseled Ubuntu images

[Chiseled Ubuntu images](#) are available for .NET 8. Chiseled images have a reduced attacked surface because they're ultra-small, have no package manager or shell, and are `non-root`. This type of image is for developers who want the benefit of appliance-style computing. Chiseled images are published to the [.NET nightly artifact registry](#).

Build multi-platform container images

Docker supports using and building [multi-platform images](#) that work across multiple environments. .NET 8 introduces a new pattern that enables you to mix and match architectures with the .NET images you build. As an example, if you're using macOS and want to target an x64 cloud service in Azure, you can build the image by using the `--platform` switch as follows:

```
docker build --pull -t app --platform linux/amd64
```

The .NET SDK now supports `$TARGETARCH` values and the `-a` argument on restore. The following code snippet shows an example:

Dockerfile

```
RUN dotnet restore -a $TARGETARCH

# Copy everything else and build app.
COPY aspnetapp/.

RUN dotnet publish -a $TARGETARCH --self-contained false --no-restore -o
/app
```

For more information, see the [Improving multi-platform container support](#) blog post.

ASP.NET composite images

As part of an effort to improve containerization performance, new ASP.NET Docker images are available that have a composite version of the runtime. This composite is built by compiling multiple MSIL assemblies into a single ready-to-run (R2R) output binary. Because these assemblies are embedded into a single image, jitting takes less time, and the startup performance of apps improves. The other big advantage of the composite over the regular ASP.NET image is that the composite images have a smaller size on disk.

There is a caveat to be aware of. Since composites have multiple assemblies embedded into one, they have tighter version coupling. Apps can't use custom versions of framework or ASP.NET binaries.

Composite images are available for the Alpine Linux, Jammy Chiseled, and Mariner Distroless platforms from the mcr.microsoft.com/dotnet/nightly/aspnet repo. The tags are listed with the `-composite` suffix on the [ASP.NET Docker page](#).

Container publishing

- [Performance and compatibility](#)
- [Authentication](#)
- [Publish to tar.gz archive](#)

Performance and compatibility

.NET 8 has improved performance for pushing containers to remote registries, especially Azure registries. Speedup comes from pushing layers in one operation and, for registries

that don't support atomic uploads, a more reliable chunking mechanism.

These improvements also mean that more registries are supported: Harbor, Artifactory, Quay.io, and Podman.

Authentication

.NET 8 adds support for OAuth token exchange authentication (Azure Managed Identity) when pushing containers to registries. This support means that you can now push to registries like Azure Container Registry without any authentication errors. The following commands show an example publishing flow:

Console

```
> az acr login -n <your registry name>
> dotnet publish -r linux-x64 -p PublishProfile=DefaultContainer
```

For more information containerizing .NET apps, see [Containerize a .NET app with dotnet publish](#).

Publish to tar.gz archive

Starting in .NET 8, you can create a container directly as a *tar.gz* archive. This feature is useful if your workflow isn't straightforward and requires that you, for example, run a scanning tool over your images before pushing them. Once the archive is created, you can move it, scan it, or load it into a local Docker toolchain.

To publish to an archive, add the `ContainerArchiveOutputPath` property to your `dotnet publish` command, for example:

.NET CLI

```
dotnet publish \
-p PublishProfile=DefaultContainer \
-p ContainerArchiveOutputPath=./images/sdk-container-demo.tar.gz
```

You can specify either a folder name or a path with a specific file name.

Source-generated COM interop

.NET 8 includes a new source generator that supports interoperating with COM interfaces. You can use the [GeneratedComInterfaceAttribute](#) to mark an interface as a

COM interface for the source generator. The source generator will then generate code to enable calling from C# code to unmanaged code. It also generates code to enable calling from unmanaged code into C#. This source generator integrates with [LibraryImportAttribute](#), and you can use types with the [GeneratedComInterfaceAttribute](#) as parameters and return types in [LibraryImport](#)-attributed methods.

```
C#  
  
using System.Runtime.InteropServices;  
using System.Runtime.InteropServices.Marshalling;  
  
[GeneratedComInterface]  
[Guid("5401c312-ab23-4dd3-aa40-3cb4b3a4683e")]  
interface IComInterface  
{  
    void DoWork();  
}  
  
internal class MyNativeLib  
{  
    [LibraryImport(nameof(MyNativeLib))]  
    public static partial void GetComInterface(out IComInterface  
comInterface);  
}
```

The source generator also supports the new [GeneratedComClassAttribute](#) attribute to enable you to pass types that implement interfaces with the [GeneratedComInterfaceAttribute](#) attribute to unmanaged code. The source generator will generate the code necessary to expose a COM object that implements the interfaces and forwards calls to the managed implementation.

Methods on interfaces with the [GeneratedComInterfaceAttribute](#) attribute support all the same types as [LibraryImportAttribute](#), and [LibraryImportAttribute](#) now supports [GeneratedComInterface](#)-attributed types and [GeneratedComClass](#)-attributed types.

If your C# code only uses a [GeneratedComInterface](#)-attributed interface to either wrap a COM object from unmanaged code or wrap a managed object from C# to expose to unmanaged code, you can use the options in the [Options](#) property to customize which code will be generated. These options mean you don't need to write marshallers for scenarios that you know won't be used.

The source generator uses the new [StrategyBasedComWrappers](#) type to create and manage the COM object wrappers and the managed object wrappers. This new type handles providing the expected .NET user experience for COM interop, while providing customization points for advanced users. If your application has its own mechanism for defining types from COM or if you need to support scenarios that source-generated

COM doesn't currently support, consider using the new [StrategyBasedComWrappers](#) type to add the missing features for your scenario and get the same .NET user experience for your COM types.

If you're using Visual Studio, new analyzers and code fixes make it easy to convert your existing COM interop code to use source-generated interop. Next to each interface that has the [ComImportAttribute](#), a lightbulb offers an option to convert to source-generated interop. The fix changes the interface to use the [GeneratedComInterfaceAttribute](#) attribute. And next to every class that implements an interface with [GeneratedComInterfaceAttribute](#), a lightbulb offers an option to add the [GeneratedComClassAttribute](#) attribute to the type. Once your types are converted, you can move your [DllImport](#) methods to use [LibraryImportAttribute](#).

Limitations

The COM source generator doesn't support apartment affinity, using the `new` keyword to activate a COM CoClass, and the following APIs:

- [IDispatch](#)-based interfaces.
- [IInspectable](#)-based interfaces.
- COM properties and events.

.NET on Linux

Minimum support baselines for Linux

The minimum support baselines for Linux have been updated for .NET 8. .NET is built targeting Ubuntu 16.04, for all architectures. That's primarily important for defining the minimum `glibc` version for .NET 8. .NET 8 will fail to start on distro versions that include an older glibc, such as Ubuntu 14.04 or Red Hat Enterprise Linux 7.

For more information, see [Red Hat Enterprise Linux Family support ↗](#).

Build your own .NET on Linux

In previous .NET versions, you could build .NET from source, but it required you to create a "source tarball" from the [dotnet/installer ↗](#) repo commit that corresponded to a release. In .NET 8, that's no longer necessary and you can build .NET on Linux directly from the [dotnet/dotnet ↗](#) repository. That repo uses [dotnet/source-build ↗](#) to build

.NET runtimes, tools, and SDKs. This is the same build that Red Hat and Canonical use to build .NET.

Building in a container is the easiest approach for most people, since the `dotnet-buildtools/prereqs` container images contain all the required dependencies. For more information, see the [build instructions ↗](#).

Cross-built Windows apps

When you build apps that target Windows on non-Windows platforms, the resulting executable is now updated with any specified Win32 resources—for example, application icon, manifest, version information.

Previously, applications had to be built on Windows in order to have such resources. Fixing this gap in cross-building support has been a popular request, as it was a significant pain point affecting both infrastructure complexity and resource usage.

AOT compilation for Android apps

To decrease app size, .NET and .NET MAUI apps that target Android use *profiled* ahead-of-time (AOT) compilation mode when they're built in Release mode. Profiled AOT compilation affects fewer methods than regular AOT compilation. .NET 8 introduces the `<AndroidStripILAAfterAOT>` property that lets you opt-in to further AOT compilation for Android apps to decrease app size even more.

XML

```
<PropertyGroup>
  <AndroidStripILAAfterAOT>true</AndroidStripILAAfterAOT>
</PropertyGroup>
```

By default, setting `AndroidStripILAAfterAOT` to `true` overrides the default `AndroidEnableProfiledAot` setting, allowing (nearly) all methods that were AOT-compiled to be trimmed. You can also use profiled AOT and IL stripping together by explicitly setting both properties to `true`:

XML

```
<PropertyGroup>
  <AndroidStripILAAfterAOT>true</AndroidStripILAAfterAOT>
  <AndroidEnableProfiledAot>true</AndroidEnableProfiledAot>
</PropertyGroup>
```

Code analysis

.NET 8 includes several new code analyzers and fixers to help verify that you're using .NET library APIs correctly and efficiently. The following table summarizes the new analyzers.

[] Expand table

| Rule ID | Category | Description |
|-------------------|-----------------|---|
| CA1856 | Performance | Fires when the ConstantExpectedAttribute attribute is not applied correctly on a parameter. |
| CA1857 | Performance | Fires when a parameter is annotated with ConstantExpectedAttribute but the provided argument isn't a constant. |
| CA1858 | Performance | To determine whether a string starts with a given prefix, it's better to call String.StartsWith than to call String.IndexOf and then compare the result with zero. |
| CA1859 | Performance | This rule recommends upgrading the type of specific local variables, fields, properties, method parameters, and method return types from interface or abstract types to concrete types when possible. Using concrete types leads to higher quality generated code. |
| CA1860 | Performance | To determine whether a collection type has any elements, it's better to use <code>Length</code> , <code>Count</code> , or <code>IsEmpty</code> than to call Enumerable.Any . |
| CA1861 | Performance | Constant arrays passed as arguments aren't reused when called repeatedly, which implies a new array is created each time. To improve performance, consider extracting the array to a static readonly field. |
| CA1865- CA1867 | Performance | The char overload is a better-performing overload for a string with a single char. |
| CA2021 | Reliability | Enumerable.Cast<TResult>(IEnumerable) and Enumerable.OfType<TResult>(IEnumerable) require compatible types to function correctly. Widening and user-defined conversions aren't supported with generic types. |
| CA1510- CA1513 | Maintainability | Throw helpers are simpler and more efficient than an <code>if</code> block constructing a new exception instance. These four analyzers were created for the following exceptions: ArgumentNullException , ArgumentException , ArgumentOutOfRangeException and ObjectDisposedException . |

Windows Presentation Foundation

- [Hardware acceleration](#)
- [OpenFileDialog](#)

Hardware acceleration

Previously, all WPF applications that were accessed remotely had to use software rendering, even if the system had hardware rendering capabilities. .NET 8 adds an option that lets you opt into hardware acceleration for Remote Desktop Protocol (RDP).

Hardware acceleration refers to the use of a computer's graphics processing unit (GPU) to speed up the rendering of graphics and visual effects in an application. This can result in improved performance and more seamless, responsive graphics. In contrast, software rendering relies solely on the computer's central processing unit (CPU) to render graphics, which can be slower and less effective.

To opt in, set the `Switch.System.Windows.Media.EnableHardwareAccelerationInRdp` configuration property to `true` in a `runtimeconfig.json` file. For more information, see [Hardware acceleration in RDP](#).

OpenFileDialog

WPF includes a new dialog box control called `OpenFileDialog`. This control lets app users browse and select folders. Previously, app developers relied on third-party software to achieve this functionality.

C#

```
var openFileDialog = new OpenFileDialog()
{
    Title = "Select folder to open ...",
    InitialDirectory = Environment.GetFolderPath(
        Environment.SpecialFolder.ProgramFiles)
};

string folderName = "";
if (openFileDialog.ShowDialog())
{
    folderName = openFileDialog.FolderName;
}
```

For more information, see [WPF File Dialog Improvements in .NET 8 \(.NET blog\)](#).

NuGet

Starting in .NET 8, NuGet verifies signed packages on Linux by default. NuGet continues to verify signed packages on Windows as well.

Most users shouldn't notice the verification. However, if you have an existing root certificate bundle located at `/etc/pki/ca-trust/extracted/pem objsign-ca-bundle.pem`, you may see trust failures accompanied by [warning NU3042](#).

You can opt out of verification by setting the environment variable

`DOTNET_NUGET_SIGNATURE_VERIFICATION` to `false`.

Diagnostics

C# Hot Reload supports modifying generics

Starting in .NET 8, C# Hot Reload [supports modifying generic types and generic methods](#). When you debug console, desktop, mobile, or WebAssembly applications with Visual Studio, you can apply changes to generic classes and generic methods in C# code or Razor pages. For more information, see the [full list of edits supported by Roslyn](#).

See also

- [Breaking changes in .NET 8](#)
- [What's new in ASP.NET Core 8.0](#)

.NET blog

- [Announcing .NET 8](#)
- [Announcing .NET 8 RC 2](#)
- [Announcing .NET 8 RC 1](#)
- [Announcing .NET 8 Preview 7](#)
- [Announcing .NET 8 Preview 6](#)
- [Announcing .NET 8 Preview 5](#)
- [Announcing .NET 8 Preview 4](#)
- [Announcing .NET 8 Preview 3](#)
- [Announcing .NET 8 Preview 2](#)
- [Announcing .NET 8 Preview 1](#)
- [ASP.NET Core in .NET 8](#)
- [ASP.NET Core updates in .NET 8 RC 2](#)

- [ASP.NET Core updates in .NET 8 RC 1](#)
- [ASP.NET Core updates in .NET 8 Preview 7](#)
- [ASP.NET Core updates in .NET 8 Preview 6](#)
- [ASP.NET Core updates in .NET 8 Preview 5](#)
- [ASP.NET Core updates in .NET 8 Preview 4](#)
- [ASP.NET Core updates in .NET 8 Preview 3](#)
- [ASP.NET Core updates in .NET 8 Preview 2](#)
- [ASP.NET Core updates in .NET 8 Preview 1](#)

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

.NET is an open source project.
Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

Breaking changes in .NET 8

Article • 12/05/2023

If you're migrating an app to .NET 8, the breaking changes listed here might affect you. Changes are grouped by technology area, such as ASP.NET Core or Windows Forms.

This article categorizes each breaking change as *binary incompatible* or *source incompatible*, or as a *behavioral change*:

- **Binary incompatible** - When run against the new runtime or component, existing binaries may encounter a breaking change in behavior, such as failure to load or execute, and if so, require recompilation.
- **Source incompatible** - When recompiled using the new SDK or component or to target the new runtime, existing source code may require source changes to compile successfully.
- **Behavioral change** - Existing code and binaries may behave differently at run time. If the new behavior is undesirable, existing code would need to be updated and recompiled.

ⓘ Note

This article is a work in progress. It's not a complete list of breaking changes in .NET 8. To query breaking changes that are still pending publication, see [Issues of .NET](#).

ASP.NET Core

↔ [Expand table](#)

| Title | Type of change |
|--|---------------------|
| ConcurrencyLimiterMiddleware is obsolete | Source incompatible |
| Custom converters for serialization removed | Behavioral change |
| ISystemClock is obsolete | Source incompatible |
| Minimal APIs: IFormFile parameters require anti-forgery checks | Behavioral change |
| Rate-limiting middleware requires AddRateLimiter | Behavioral change |

| Title | Type of change |
|--|---------------------|
| Security token events return a JsonWebToken | Behavioral change |
| TrimMode defaults to full for Web SDK projects | Source incompatible |

Containers

[\[\] Expand table](#)

| Title | Type of change |
|---|---------------------------------------|
| 'ca-certificates' and 'krb5-libs' packages removed from Alpine images | Binary incompatible |
| Debian container images upgraded to Debian 12 | Binary incompatible/behavioral change |
| Default ASP.NET Core port changed to 8080 | Behavioral change |
| 'libintl' package removed from Alpine images | Behavioral change |
| Multi-platform container tags are Linux-only | Behavioral change |
| New 'app' user in Linux images | Behavioral change |

Core .NET libraries

[\[\] Expand table](#)

| Title | Type of change |
|---|---------------------|
| Activity operation name when null | Behavioral change |
| AnonymousPipeServerStream.Dispose behavior | Behavioral change |
| API obsoletions with custom diagnostic IDs | Source incompatible |
| Backslash mapping in Unix file paths | Behavioral change |
| Base64.DecodeFromUtf8 methods ignore whitespace | Behavioral change |
| Boolean-backed enum type support removed | Behavioral change |
| Enumerable.Sum throws new OverflowException for some inputs | Behavioral change |
| FileStream writes when pipe is closed | Behavioral change |

| Title | Type of change |
|---|---------------------|
| GC.GetGeneration might return Int32.MaxValue | Behavioral change |
| GetFolderPath behavior on Unix | Behavioral change |
| GetSystemVersion no longer returns ImageRuntimeVersion | Behavioral change |
| ITypeDescriptorContext nullable annotations | Source incompatible |
| Legacy Console.ReadKey removed | Behavioral change |
| Method builders generate parameters with HasDefaultValue set to false | Behavioral change |
| ProcessStartInfo.WindowStyle honored when UseShellExecute is false | Behavioral change |
| Runtimeldentifier returns platform for which runtime was built | Behavioral change |

Cryptography

[\[\] Expand table](#)

| Title | Type of change | Introduced |
|--|---------------------|------------|
| AesGcm authentication tag size on macOS | Behavioral change | Preview 1 |
| RSA.EncryptValue and RSA.DecryptValue obsolete | Source incompatible | Preview 1 |

Deployment

[\[\] Expand table](#)

| Title | Type of change |
|--|---------------------------------------|
| Host determines RID-specific assets | Binary incompatible/behavioral change |
| .NET Monitor only includes distroless images | Behavioral change |
| StripSymbols defaults to true | Behavioral change |

Entity Framework Core

[Breaking changes in EF Core 8](#)

Extensions

[Expand table](#)

| Title | Type of change |
|---|---------------------|
| ActivatorUtilities.CreateInstance behaves consistently | Behavioral change |
| ActivatorUtilities.CreateInstance requires non-null provider | Behavioral change |
| ConfigurationBinder throws for mismatched value | Behavioral change |
| ConfigurationManager package no longer references System.Security.Permissions | Source incompatible |
| DirectoryServices package no longer references System.Security.Permissions | Source incompatible |
| Empty keys added to dictionary by configuration binder | Behavioral change |
| HostApplicationBuilderSettings.Args respected by HostApplicationBuilder constructor | Behavioral change |
| ManagementDateTimeConverter.ToDateTime returns a local time | Behavioral change |
| System.Formats.Cbor DateTimeOffset formatting change | Behavioral change |

Globalization

[Expand table](#)

| Title | Type of change |
|---|-------------------|
| Date and time converters honor culture argument | Behavioral change |
| TwoDigitYearMax default is 2049 | Behavioral change |

Interop

[Expand table](#)

| Title | Type of change |
|--|---------------------|
| CreateObjectFlags.Unwrap only unwraps on target instance | Behavioral change |
| Custom marshallers require additional members | Source incompatible |

| Title | Type of change |
|---|---------------------|
| IDispatchImplAttribute API is removed | Binary incompatible |
| JSFunctionBinding implicit public default constructor removed | Binary incompatible |
| SafeHandle types must have public constructor | Source incompatible |

Networking

[\[+\] Expand table](#)

| Title | Type of change |
|--|-------------------|
| SendFile throws NotSupportedException for connectionless sockets | Behavioral change |

Reflection

[\[+\] Expand table](#)

| Title | Type of change |
|--|-------------------|
| IntPtr no longer used for function pointer types | Behavioral change |

SDK

[\[+\] Expand table](#)

| Title | Type of change |
|--|--|
| CLI console output uses UTF-8 | Behavioral change/Source and binary incompatible |
| Console encoding not UTF-8 after completion | Behavioral change/Binary incompatible |
| Containers default to use the 'latest' tag | Behavioral change |
| 'dotnet pack' uses Release configuration | Behavioral change/Source incompatible |
| 'dotnet publish' uses Release configuration | Behavioral change/Source incompatible |
| Implicit using for System.Net.Http no longer added | Behavioral change/Source incompatible |
| MSBuild custom derived build events deprecated | Behavioral change |

| Title | Type of change |
|---|---------------------------------------|
| MSBuild respects DOTNET_CLI_UI_LANGUAGE | Behavioral change |
| Runtime-specific apps not self-contained | Source/binary incompatible |
| --arch option doesn't imply self-contained | Behavioral change |
| 'dotnet restore' produces security vulnerability warnings | Behavioral change |
| SDK uses a smaller RID graph | Behavioral change/Source incompatible |
| Source Link included in the .NET SDK | Source incompatible |
| Trimming may not be used with .NET Standard or .NET Framework | Behavioral change |
| Unlisted packages not installed by default for .NET tools | Behavioral change |
| Version requirements for .NET 8 SDK | Source incompatible |

Serialization

[\[+\] Expand table](#)

| Title | Type of change |
|---|-------------------|
| BinaryFormatter disabled for most projects | Behavioral change |
| PublishedTrimmed projects fail reflection-based serialization | Behavioral change |
| Reflection-based deserializer resolves metadata eagerly | Behavioral change |

Windows Forms

[\[+\] Expand table](#)

| Title | Type of change |
|--|-------------------|
| Anchor layout changes | Behavioral change |
| DefaultValueAttribute removed from some properties | Behavioral change |
| ExceptionCollection ctor throws ArgumentException | Behavioral change |

| Title | Type of change |
|---|---------------------|
| Forms scale according to AutoScaleMode | Behavioral change |
| ImageList.ColorDepth default is Depth32Bit | Behavioral change |
| System.Windows.Extensions doesn't reference System.Drawing.Common | Source incompatible |
| TableLayoutStyleCollection throws ArgumentException | Behavioral change |
| Top-level forms scale minimum and maximum size to DPI | Behavioral change |
| WFDEV002 obsoletion is now an error | Source incompatible |

See also

- [What's new in .NET 8](#)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

.NET is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

What's new in .NET 7

Article • 03/09/2023

.NET 7 is the successor to [.NET 6](#) and focuses on being unified, modern, simple, and *fast*. .NET 7 will be [supported for 18 months](#) as a standard-term support (STS) release (previously known as a *current* release).

This article lists the new features of .NET 7 and provides links to more detailed information on each.

To find all the .NET articles that have been updated for .NET 7, see [.NET docs: What's new for the .NET 7 release](#).

Performance

Performance is a key focus of .NET 7, and all of its features are designed with performance in mind. In addition, .NET 7 includes the following enhancements aimed purely at performance:

- On-stack replacement (OSR) is a complement to tiered compilation. It allows the runtime to change the code executed by a currently running method in the middle of its execution (that is, while it's "on stack"). Long-running methods can switch to more optimized versions mid-execution.
- Profile-guided optimization (PGO) now works with OSR and is easier to enable (by adding `<TieredPGO>true</TieredPGO>` to your project file). PGO can also instrument and optimize additional things, such as delegates.
- Improved code generation for Arm64.
- [Native AOT](#) produces a standalone executable in the target platform's file format with no external dependencies. It's entirely native, with no IL or JIT, and provides fast startup time and a small, self-contained deployment. In .NET 7, Native AOT focuses on console apps and requires apps to be trimmed.
- Performance improvements to the Mono runtime, which powers Blazor WebAssembly, Android, and iOS apps.

For a detailed look at many of the performance-focused features that make .NET 7 so fast, see the [Performance improvements in .NET 7](#) blog post.

System.Text.Json serialization

.NET 7 includes improvements to System.Text.Json serialization in the following areas:

- **Contract customization** gives you more control over how types are serialized and deserialized. For more information, see [Customize a JSON contract](#).
- **Polymorphic serialization** for user-defined type hierarchies. For more information, see [Serialize properties of derived classes](#).
- Support for **required members**, which are properties that must be present in the JSON payload for deserialization to succeed. For more information, see [Required properties](#).

For information about these and other updates, see the [What's new in System.Text.Json in .NET 7](#) blog post.

Generic math

.NET 7 and C# 11 include innovations that allow you to perform mathematical operations generically—that is, without having to know the exact type you're working with. For example, if you wanted to write a method that adds two numbers, previously you had to add an overload of the method for each type. Now you can write a single, generic method, where the type parameter is constrained to be a number-like type. For more information, see the [Generic math](#) article and the [Generic math](#) blog post.

Regular expressions

.NET's [regular expression](#) library has seen significant functional and performance improvements in .NET 7:

- The new option [RegexOptions.NonBacktracking](#) enables matching using an approach that avoids backtracking and guarantees linear-time processing in the length of the input. The nonbacktracking engine can't be used in a right-to-left search and has a few other restrictions, but is fast for all regular expressions and inputs. For more information, see [Nonbacktracking mode](#).
- Regular expression source generators are new. Source generators build an engine that's optimized for *your* pattern at compile time, providing throughput performance benefits. The source that's emitted is part of your project, so you can view and debug it. In addition, a new source-generator diagnostic `SYSLIB1045` alerts you to places you use [Regex](#) that could be converted to the source generator. For more information, see [.NET regular expression source generators](#).
- For case-insensitive searches, .NET 7 includes large performance gains. The gains come because specifying [RegexOptions.IgnoreCase](#) no longer calls [ToLower](#) on each character in the pattern and on each character in the input. Instead, all casing-related work is done when the [Regex](#) is constructed.

- `Regex` now supports spans for some APIs. The following new methods have been added as part of this support:
 - `Regex.EnumerateMatches`
 - `Regex.Count`
 - `Regex.IsMatch(ReadOnlySpan<Char>)` (and a few other overloads)

For more information about these and other improvements, see the [Regular expression improvements in .NET 7](#) blog post.

.NET libraries

Many improvements have been made to .NET library APIs. Some are mentioned in other, dedicated sections of this article. Some others are summarized in the following table.

[Expand table](#)

| Description | APIs | Further information |
|---|---|--|
| Support for microseconds and nanoseconds in <code>TimeSpan</code> , <code>TimeOnly</code> , <code>DateTime</code> , and <code>DateTimeOffset</code> types | <ul style="list-style-type: none"> - <code>DateTime.Microsecond</code> - <code>DateTime.Nanosecond</code> - <code>DateTime.AddMicroseconds(Double)</code> - New <code>DateTime</code> constructor overloads - <code>DateTimeOffset.Microsecond</code> - <code>DateTimeOffset.Nanosecond</code> - <code>DateTimeOffset.AddMicroseconds(Double)</code> - New <code>DateTimeOffset</code> constructor overloads - <code>TimeOnly.Microsecond</code> - <code>TimeOnly.Nanosecond</code> - <code>TimeSpan.Microseconds</code> - <code>TimeSpan.Nanoseconds</code> - <code>TimeSpan.FromMicroseconds(Double)</code> - And others... | These APIs mean you no longer have to perform computations on the "tick" value to determine microsecond and nanosecond values. For more information, see the .NET 7 Preview 4 blog post. |
| APIs for reading, writing, archiving, and extracting Tar archives | <code>System.Formats.Tar</code> | For more information, see the .NET 7 Preview 4 and .NET 7 Preview 6 blog posts. |
| Rate limiting APIs to protect a resource by | <code>RateLimiter</code> and others in the <code>System.Threading.RateLimiting</code> NuGet package | For more information, see Rate limit an HTTP handler in .NET and Announcing rate limiting for .NET . |

| Description | APIs | Further information |
|--|--|--|
| keeping traffic at a safe level | | |
| APIs to read <i>all</i> the data from a Stream | <ul style="list-style-type: none"> - Stream.ReadExactly - Stream.ReadAtLeast | Stream.Read may return less data than what's available in the stream. The new <code>ReadExactly</code> methods read <i>exactly</i> the number of bytes requested, and the new <code>ReadAtLeast</code> methods read <i>at least</i> the number of bytes requested. For more information, see the .NET 7 Preview 5 blog post. |
| New type converters for <code>DateOnly</code> , <code>TimeOnly</code> , <code>Int128</code> , <code>UInt128</code> , and <code>Half</code> | <p>In the System.ComponentModel namespace:</p> <ul style="list-style-type: none"> - DateOnlyConverter - TimeOnlyConverter - Int128Converter - UInt128Converter - HalfConverter | Type converters are often used to convert value types to and from a string. These new APIs add type converters for types that were added more recently. |
| Metrics support for IMemoryCache | <ul style="list-style-type: none"> - MemoryCacheStatistics - MemoryCache.GetCurrentStatistics() | GetCurrentStatistics() lets you use event counters or metrics APIs to track statistics for one or more memory caches. For more information, see the .NET 7 Preview 4 blog post. |
| APIs to get and set Unix file permissions | <ul style="list-style-type: none"> - System.IO.Unix FileMode enum - File.GetUnixFileMode - File.SetUnixFileMode - FileSystemInfo.Unix FileMode - Directory.CreateDirectory(String, Unix FileMode) - FileStreamOptions.UnixCreateMode | For more information, see the .NET 7 Preview 7 blog post. |
| Attribute to indicate what kind of syntax is expected in a string | StringSyntaxAttribute | For example, you can specify that a <code>string</code> parameter expects a regular expression by attributing the parameter with <code>[StringSyntax(StringSyntaxAttribute.Regex)]</code> . |
| APIs to interop with JavaScript when running in the browser or other WebAssembly architectures | System.Runtime.InteropServices.JavaScript | JavaScript apps can use the expanded WebAssembly support in .NET 7 to reuse .NET libraries from JavaScript. For more information, see Use .NET from any JavaScript app in .NET 7 . |

Observability

.NET 7 makes improvements to *observability*. Observability helps you understand the state of your app as it scales and as the technical complexity increases. .NET's observability implementation is primarily built around [OpenTelemetry](#). Improvements include:

- The new `Activity.CurrentChanged` event, which you can use to detect when the span context of a managed thread changes.
- New, performant enumerator methods for `Activity` properties: `EnumerateTagObjects()`, `EnumerateLinks()`, and `EnumerateEvents()`.

For more information, see the [.NET 7 Preview 4](#) blog post.

.NET SDK

The .NET 7 [SDK](#) improves the CLI template experience. It also enables publishing to containers, and central package management with NuGet.

Templates

Some welcome improvements have been made to the `dotnet new` command and to template authoring.

`dotnet new`

The `dotnet new` CLI command, which creates a new project, configuration file, or solution based on a template, now supports [tab completion](#) for exploring:

- Available template names
- Template options
- Allowable option values

In addition, for better conformity, the `install`, `uninstall`, `search`, `list`, and `update` subcommands no longer have the `--` prefix.

Authoring

Template *constraints*, a new concept for .NET 7, let you define the context in which your templates are allowed. Constraints help the template engine determine which templates it should show in commands like `dotnet new list`. You can constrain your template to an operating system, a template engine host (for example, the .NET CLI or New Project dialog in Visual Studio), and an installed workload. You define constraints in your template's configuration file.

Also in the template configuration file, you can now annotate a template parameter as allowing multiple values. For example, the [web template](#) allows multiple forms of authentication.

For more information, see the [.NET 7 Preview 6](#) blog post.

Publish to a container

Containers are one of the easiest ways to distribute and run a wide variety of applications and services in the cloud. Container images are now a supported output type of the .NET SDK, and you can create containerized versions of your applications using [dotnet publish](#). For more information about the feature, see [Announcing built-in container support for the .NET SDK](#). For a tutorial, see [Containerize a .NET app with dotnet publish](#).

Central package management

You can now manage common dependencies in your projects from one location using NuGet's central package management (CPM) feature. To enable it, you add a `Directory.Packages.props` file to the root of your repository. In this file, set the MSBuild property `ManagePackageVersionsCentrally` to `true` and add versions for common package dependency using `PackageVersion` items. Then, in the individual project files, you can omit `Version` attributes from any `PackageReference` items that refer to centrally managed packages.

For more information, see [Central package management](#).

P/Invoke source generation

.NET 7 introduces a source generator for platform invokes (P/Invokes) in C#. The source generator looks for `LibraryImportAttribute` on `static`, `partial` methods to trigger compile-time source generation of marshalling code. By generating the marshalling code at compile time, no IL stub needs to be generated at run time, as it does when using `DllImportAttribute`. The source generator improves application performance and also allows the app to be ahead-of-time (AOT) compiled. For more information, see [Source generation for platform invokes](#) and [Use custom marshallsers in source-generated P/Invokes](#).

Related releases

This section contains information about related products that have releases that coincide with the .NET 7 release.

Visual Studio 2022 version 17.4

For more information, see [What's new in Visual Studio 2022](#).

C# 11

C# 11 includes support for [generic math](#), raw string literals, file-scoped types, and other new features. For more information, see [What's new in C# 11](#).

F# 7

F# 7 continues the journey to make the language simpler and improve performance and interop with new C# features. For more information, see [Announcing F# 7](#).

.NET MAUI

.NET Multi-platform App UI (.NET MAUI) is a cross-platform framework for creating native mobile and desktop apps with C# and XAML. It unifies Android, iOS, macOS, and Windows APIs into a single API. For information about the latest updates, see [What's new in .NET MAUI for .NET 7](#).

ASP.NET Core

ASP.NET Core 7.0 includes rate-limiting middleware, improvements to minimal APIs, and gRPC JSON transcoding. For information about all the updates, see [What's new in ASP.NET Core 7](#).

EF Core

Entity Framework Core 7.0 includes provider-agnostic support for JSON columns, improved performance for saving changes, and custom reverse engineering templates. For information about all the updates, see [What's new in EF Core 7.0](#).

Windows Forms

Much work has gone into Windows Forms for .NET 7. Improvements have been made in the following areas:

- Accessibility
- High DPI and scaling
- Databinding

For more information, see [What's new in Windows Forms in .NET 7](#).

WPF

WPF in .NET 7 includes numerous bug fixes as well as performance and accessibility improvements. For more information, see the [What's new for WPF in .NET 7](#) blog post.

Orleans

Orleans is a cross-platform framework for building robust, scalable distributed applications. For information about the latest updates for Orleans, see [Migrate from Orleans 3.x to 7.0](#).

.NET Upgrade Assistant and CoreWCF

The .NET Upgrade Assistant now supports upgrading server-side WCF apps to [CoreWCF](#), which is a community-created port of WCF to .NET (Core). For more information, see [Upgrade a WCF server-side project to use CoreWCF](#).

ML.NET

ML.NET now includes a text classification API that makes it easy to train custom text classification models using the latest state-of-the-art deep learning techniques. For more information, see the [What's new with AutoML and tooling](#) and [Introducing the ML.NET Text Classification API](#) blog posts.

See also

- [Release notes for .NET 7](#)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



.NET feedback

.NET is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

Breaking changes in .NET 7

Article • 07/28/2023

If you're migrating an app to .NET 7, the breaking changes listed here might affect you. Changes are grouped by technology area, such as ASP.NET Core or Windows Forms.

This article indicates whether each breaking change is *binary compatible* or *source compatible*:

- **Binary compatible** - Existing binaries will load and execute successfully without recompilation, and the run-time behavior won't change.
- **Source compatible** - Source code will compile successfully without changes when targeting the new runtime or using the new SDK or component.

ASP.NET Core

[] Expand table

| Title | Binary compatible | Source compatible |
|---|-------------------|-------------------|
| API controller actions try to infer parameters from DI | ✓ | ✗ |
| ASP.NET-prefixed environment variable precedence | ✓ | ✓ |
| AuthenticateAsync for remote auth providers | ✓ | ✗ |
| Authentication in WebAssembly apps | ✗ | ✓ |
| Default authentication scheme | ✗ | ✓ |
| Event IDs for some Microsoft.AspNetCore.Mvc.Core log messages changed | ✗ | ✓ |
| Fallback file endpoints | ✗ | ✓ |
| IHubClients and IHubCallerClients hide members | ✓ | ✗ |
| Kestrel: Default HTTPS binding removed | ✗ | ✓ |
| Microsoft.AspNetCore.Server.Kestrel.Transport.Libuv and libuv.dll removed | ✗ | ✗ |
| Microsoft.Data.SqlClient updated to 4.0.1 | ✓ | ✗ |
| Middleware no longer defers to endpoint with null request | ✗ | ✓ |

| Title | Binary compatible | Source compatible |
|---|-------------------|-------------------|
| delegate | | |
| MVC's detection of an empty body in model binding changed | ✗ | ✓ |
| Output caching API changes | ✗ | ✗ |
| SignalR Hub methods try to resolve parameters from DI | ✓ | ✗ |

Core .NET libraries

[+] Expand table

| Title | Binary compatible | Source compatible |
|---|-------------------|-------------------|
| API obsolesions with default diagnostic ID | ✓ | ✗ |
| API obsolesions with non-default diagnostic IDs | ✓ | ✗ |
| BinaryFormatter serialization APIs produce compiler errors | ✓ | ✗ |
| BrotliStream no longer allows undefined CompressionLevel values | ✗ | ✓ |
| C++/CLI projects in Visual Studio | ✓ | ✗ |
| Changes to reflection invoke API exceptions | ✗ | ✓ |
| Collectible Assembly in non-collectible AssemblyLoadContext | ✗ | ✓ |
| DateTime addition methods precision change | ✓ | ✓ |
| Equals method behavior change for NaN | ✗ | ✓ |
| EventSource callback behavior | ✓ | ✓ |
| Generic type constraint on PatternContext<T> | ✗ | ✗ |
| Legacy FileStream strategy removed | ✗ | ✓ |
| Library support for older frameworks | ✗ | ✗ |
| Maximum precision for numeric format strings | ✗ | ✓ |
| Regex patterns with ranges corrected | ✓ | ✓ |
| SerializationFormat.Binary is obsolete | ✗ | ✗ |

| Title | Binary compatible | Source compatible |
|--|-------------------|-------------------|
| System.Drawing.Common config switch removed | ✓ | ✓ |
| System.Runtime.CompilerServices.Unsafe NuGet package | ✓ | ✓ |
| Time fields on symbolic links | ✗ | ✓ |
| Tracking linked cache entries | ✗ | ✓ |
| Validate CompressionLevel for BrotliStream | ✗ | ✓ |

Configuration

[\[+\] Expand table](#)

| Title | Binary compatible | Source compatible |
|--|-------------------|-------------------|
| System.diagnostics entry in app.config | ✗ | ✓ |

Cryptography

[\[+\] Expand table](#)

| Title | Binary compatible | Source compatible |
|---|-------------------|-------------------|
| Decrypting EnvelopedCms doesn't double unwrap | ✗ | ✓ |
| Dynamic X509ChainPolicy verification time | ✗ | ✓ |
| X500DistinguishedName parsing of friendly names | ✗ | ✓ |

Deployment

[\[+\] Expand table](#)

| Title | Binary compatible | Source compatible |
|-----------------------------------|-------------------|-------------------|
| All assemblies trimmed by default | ✓ | ✗ |
| Multi-level lookup is disabled | ✗ | ✓ |
| x86 host path on 64-bit Windows | ✓ | ✓ |

| Title | Binary compatible | Source compatible |
|------------------------------------|-------------------|-------------------|
| TrimmerDefaultAction is deprecated | ✓ | ✗ |

Entity Framework Core

[Breaking changes in EF Core 7](#)

Extensions

[Expand table](#)

| Title | Binary compatible | Source compatible |
|--|-------------------|-------------------|
| Binding config to dictionary extends values | ✓ | ✓ |
| ContentRootPath for apps launched by Windows Shell | ✗ | ✓ |
| Environment variable prefixes | ✗ | ✓ |

Globalization

[Expand table](#)

| Title | Binary compatible | Source compatible |
|--|-------------------|-------------------|
| Globalization APIs use ICU libraries on Windows Server | ✗ | ✓ |

Interop

[Expand table](#)

| Title | Binary compatible | Source compatible |
|--|-------------------|-------------------|
| RuntimelInformation.OSArchitecture under emulation | ✗ | ✓ |

.NET MAUI

[Expand table](#)

| Title | Binary compatible | Source compatible |
|---|-------------------|-------------------|
| Constructors accept base interface instead of concrete type | ✗ | ✓ |
| Flow direction helper methods removed | ✗ | ✗ |
| New UpdateBackground parameter | ✗ | ✓ |
| ScrollToRequest property renamed | ✗ | ✗ |
| Some Windows APIs are removed | ✗ | ✗ |

Networking

[Expand table](#)

| Title | Binary compatible | Source compatible |
|--|-------------------|-------------------|
| AllowRenegotiation default is false | ✗ | ✗ |
| Custom ping payloads on Linux | ✗ | ✓ |
| Socket.End methods don't throw ObjectDisposedException | ✗ | ✓ |

SDK and MSBuild

[Expand table](#)

| Title | Binary compatible | Source compatible |
|--|-------------------|-------------------|
| Automatic RuntimeIdentifier for certain projects | ✓ | ✗ |
| Automatic RuntimeIdentifier for publish only | ✗ | ✗ |
| CLI console output uses UTF-8 | ✗ | ✗ |
| Console encoding not UTF-8 after completion | ✗ | ✓ |
| MSBuild serialization of custom types in .NET 7 | ✗ | ✗ |

| Title | Binary compatible | Source compatible |
|---|-------------------|-------------------|
| Side-by-side SDK installations | ✗ | ✗ |
| Tool manifests in root folder | ✓ | ✓ |
| Version requirements for .NET 7 SDK | ✓ | ✓ |
| dotnet test: switch -a to alias --arch instead of --test-adapter-path ↗ | ✗ | ✗ |
| dotnet test: switch -r to alias --runtime instead of --results-dir ↗ | ✗ | ✗ |
| --output option no longer is valid for solution-level commands | ✗ | ✗ |
| SDK no longer calls ResolvePackageDependencies | ✓ | ✗ |

Serialization

[+] Expand table

| Title | Binary compatible | Source compatible |
|---|-------------------|-------------------|
| DataContractSerializer retains sign when deserializing -0 | ✗ | ✓ |
| Deserialize Version type with leading or trailing whitespace | ✗ | ✓ |
| JsonSerializerOptions copy constructor includes JsonSerializerContext | ✗ | ✓ |
| Polymorphic serialization for object types | ✗ | ✓ |
| System.Text.Json source generator fallback | ✗ | ✓ |

Windows Forms

[+] Expand table

| Title | Binary compatible | Source compatible |
|--------------------------|-------------------|-------------------|
| Obsoletions and warnings | ✓ | ✗ |

| Title | Binary compatible | Source compatible |
|---------------------------------------|-------------------|-------------------|
| Some APIs throw ArgumentNullException | ✗ | ✓ |

XML and XSLT

[Expand table](#)

| Title | Binary compatible | Source compatible |
|-------------------------------|-------------------|-------------------|
| XmlSecureResolver is obsolete | ✗ | ✗ |

See also

- [What's new in .NET 7](#)

 Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

.NET is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

What's new in .NET 6

Article • 05/26/2023

.NET 6 delivers the final parts of the .NET unification plan that started with [.NET 5](#). .NET 6 unifies the SDK, base libraries, and runtime across mobile, desktop, IoT, and cloud apps. In addition to this unification, the .NET 6 ecosystem offers:

- **Simplified development:** Getting started is easy. New language features in [C# 10](#) reduce the amount of code you need to write. And investments in the web stack and minimal APIs make it easy to quickly write smaller, faster microservices.
- **Better performance:** .NET 6 is the fastest full stack web framework, which lowers compute costs if you're running in the cloud.
- **Ultimate productivity:** .NET 6 and [Visual Studio 2022](#) provide hot reload, new git tooling, intelligent code editing, robust diagnostics and testing tools, and better team collaboration.

.NET 6 will be [supported for three years](#) as a long-term support (LTS) release.

Preview features are disabled by default. They are also not supported for use in production and may be removed in a future version. The new [RequiresPreviewFeaturesAttribute](#) is used to annotate preview APIs, and a corresponding analyzer alerts you if you're using these preview APIs.

.NET 6 is supported by Visual Studio 2022 and Visual Studio 2022 for Mac (and later versions).

This article does not cover all of the new features of .NET 6. To see all of the new features, and for further information about the features listed in this article, see the [Announcing .NET 6](#) blog post.

Performance

.NET 6 includes numerous performance improvements. This section lists some of the improvements—in [FileStream](#), [profile-guided optimization](#), and [AOT compilation](#). For detailed information, see the [Performance improvements in .NET 6](#) blog post.

FileStream

The [System.IO.FileStream](#) type has been rewritten for .NET 6 to provide better performance and reliability on Windows. Now, [FileStream](#) never blocks when created for

asynchronous I/O on Windows. For more information, see the [File IO improvements in .NET 6](#) blog post.

Profile-guided optimization

Profile-guided optimization (PGO) is where the JIT compiler generates optimized code in terms of the types and code paths that are most frequently used. .NET 6 introduces *dynamic* PGO. Dynamic PGO works hand-in-hand with tiered compilation to further optimize code based on additional instrumentation that's put in place during tier 0. Dynamic PGO is disabled by default, but you can enable it with the `DOTNET_TieredPGO` environment variable. For more information, see [JIT performance improvements](#).

Crossgen2

.NET 6 introduces Crossgen2, the successor to Crossgen, which has been removed. Crossgen and Crossgen2 are tools that provide ahead-of-time (AOT) compilation to improve the startup time of an app. Crossgen2 is written in C# instead of C++, and can perform analysis and optimization that weren't possible with the previous version. For more information, see [Conversation about Crossgen2](#).

Arm64 support

The .NET 6 release includes support for macOS Arm64 (or "Apple Silicon") and Windows Arm64 operating systems, for both native Arm64 execution and x64 emulation. In addition, the x64 and Arm64 .NET installers now install side by side. For more information, see [.NET Support for macOS 11 and Windows 11 for Arm64 and x64](#).

Hot reload

Hot reload is a feature that lets you modify your app's source code and instantly apply those changes to your running app. The feature's purpose is to increase your productivity by avoiding app restarts between edits. Hot reload is available in Visual Studio 2022 and the `dotnet watch` command-line tool. Hot reload works with most types of .NET apps, and for C#, Visual Basic, and C++ source code. For more information, see the [Hot reload blog post](#).

.NET MAUI

.NET Multi-platform App UI (.NET MAUI) is still in *preview*, with a release candidate coming in the first quarter of 2022 and general availability (GA) in the second quarter of 2022. .NET MAUI makes it possible to build native client apps for desktop and mobile operating systems with a single codebase. For more information, see the [Update on .NET Multi-platform App UI](#) blog post.

C# 10 and templates

C# 10 includes innovations such as `global using` directives, file-scoped namespace declarations, and record structs. For more information, see [What's new in C# 10](#).

In concert with that work, the .NET SDK project templates for C# have been modernized to use some of the new language features:

- `async Main` method
- Top-level statements
- Target-typed new expressions
- [Implicit global using directives](#)
- File-scoped namespaces
- Nullable reference types

By adding these new language features to the project templates, new code starts with the features enabled. However, existing code isn't affected when you upgrade to .NET 6. For more information about these template changes, see the [.NET SDK: C# project templates modernized](#) blog post.

F# and Visual Basic

F# 6 adds several improvements to the F# language and F# Interactive. For more information, see [What's new in F# 6](#).

Visual Basic has improvements in the Visual Studio experience and Windows Forms project startup.

SDK Workloads

To keep the size of the .NET SDK smaller, some components have been placed in new, optional *SDK workloads*. These components include .NET MAUI and Blazor WebAssembly AOT. If you use Visual Studio, it will take care of installing any SDK workloads that you need. If you use the [.NET CLI](#), you can manage workloads using the new `dotnet workload` commands:

| Command | Description |
|---|---|
| dotnet workload search | Searches for available workloads. |
| dotnet workload install | Installs a specified workload. |
| dotnet workload uninstall | Removes a specified workload. |
| dotnet workload update | Updates installed workloads. |
| dotnet workload repair | Reinstalls all installed workloads to repair a broken installation. |
| dotnet workload list | Lists installed workloads. |

For more information, see [Optional SDK workloads ↗](#).

System.Text.Json APIs

Many improvements have been made in [System.Text.Json](#) in .NET 6, such that it is now an "industrial strength" serialization solution.

Source generator

.NET 6 adds a new [source generator](#) for [System.Text.Json](#). Source generation works with [JsonSerializer](#) and can be configured in multiple ways. It can improve performance, reduce memory usage, and facilitate assembly trimming. For more information, see [How to choose reflection or source generation in System.Text.Json](#) and [How to use source generation in System.Text.Json](#).

Writeable DOM

A new, writeable document object model (DOM) has been added, which supplements the pre-existing read-only DOM. The new API provides a lightweight serialization alternative for cases when use of plain old CLR object (POCO) types isn't possible. It also allows you to efficiently navigate to a subsection of a large JSON tree and read an array or deserialize a POCO from that subsection. The following new types have been added to support the writeable DOM:

- [JsonNode](#)
- [JsonArray](#)
- [JsonObject](#)
- [JsonValue](#)

For more information, see [JSON DOM choices](#).

IAsyncEnumerable serialization

[System.Text.Json](#) now supports serialization and deserialization with `IAsyncEnumerable<T>` instances. Asynchronous serialization methods enumerate any `IAsyncEnumerable<T>` instances in an object graph and then serialize them as JSON arrays. For deserialization, the new method `JsonSerializer.DeserializeAsyncEnumerable< TValue >(Stream, JsonSerializerOptions, CancellationToken)` was added. For more information, see [IAsyncEnumerable serialization](#).

Other new APIs

New serialization interfaces for validation and defaulting values:

- [IJsonOnDeserialized](#)
- [IJsonOnDeserializing](#)
- [IJsonOnSerialized](#)
- [IJsonOnSerializing](#)

For more information, see [Callbacks](#).

New property ordering attribute:

- [JsonPropertyOrderAttribute](#)

For more information, see [Configure the order of serialized properties](#).

New method to write "raw" JSON:

- [Utf8JsonWriter.WriteRawValue](#)

For more information, see [Write Raw JSON](#).

Synchronous serialization and deserialization to a stream:

- [JsonSerializer.Deserialize\(Stream, Type, JsonSerializerOptions\)](#)
- [JsonSerializer.Deserialize\(Stream, Type, JsonSerializerContext\)](#)
- [JsonSerializer.Deserialize< TValue >\(Stream, JsonSerializerOptions\)](#)
- [JsonSerializer.Deserialize< TValue >\(Stream, JsonTypeInfo< TValue >\)](#)
- [JsonSerializer.Serialize\(Stream, Object, Type, JsonSerializerOptions\)](#)
- [JsonSerializer.Serialize\(Stream, Object, Type, JsonSerializerContext\)](#)
- [JsonSerializer.Serialize< TValue >\(Stream, TValue, JsonSerializerOptions\)](#)
- [JsonSerializer.Serialize< TValue >\(Stream, TValue, JsonTypeInfo< TValue >\)](#)

New option to ignore an object when a reference cycle is detected during serialization:

- `ReferenceHandler.IgnoreCycles`

For more information, see [Ignore circular references](#).

For more information about serializing and deserializing with `System.Text.Json`, see [JSON serialization and deserialization in .NET](#).

HTTP/3

.NET 6 includes preview support for HTTP/3, a new version of HTTP. HTTP/3 solves some existing functional and performance challenges by using a new underlying connection protocol called QUIC. QUIC establishes connections more quickly, and connections are independent of the IP address, allowing mobile clients to roam between Wi-fi and cellular networks. For more information, see [Use HTTP/3 with HttpClient](#).

ASP.NET Core

ASP.NET Core includes improvements in minimal APIs, ahead-of-time (AOT) compilation for Blazor WebAssembly apps, and single-page apps. In addition, Blazor components can now be rendered from JavaScript and integrated with existing JavaScript based apps. For more information, see [What's new in ASP.NET Core 6](#).

OpenTelemetry

.NET 6 brings improved support for [OpenTelemetry](#), which is a collection of tools, APIs, and SDKs that help you analyze your software's performance and behavior. APIs in the `System.Diagnostics.Metrics` namespace implement the [OpenTelemetry Metrics API specification](#). For example, there are four instrument classes to support different metrics scenarios. The instrument classes are:

- `Counter<T>`
- `Histogram<T>`
- `ObservableCounter<T>`
- `ObservableGauge<T>`

Security

.NET 6 adds preview support for two key security mitigations: Control-flow Enforcement Technology (CET) and "write exclusive execute" (W^X).

CET is an Intel technology available in some newer Intel and AMD processors. It adds capabilities to the hardware that protect against some control-flow hijacking attacks. .NET 6 provides support for CET for Windows x64 apps, and you must explicitly enable it. For more information, see [.NET 6 compatibility with Intel CET shadow stacks](#).

W[^]X is available all operating systems with .NET 6 but only enabled by default on Apple Silicon. W[^]X blocks the simplest attack path by disallowing memory pages to be writeable and executable at the same time.

IL trimming

Trimming of self-contained deployments is improved. In .NET 5, only unused assemblies were trimmed. .NET 6 adds trimming of unused types and members too. In addition, trim warnings, which alert you to places where trimming may remove code that's used at run time, are now *enabled* by default. For more information, see [Trim self-contained deployments and executables](#).

Code analysis

The .NET 6 SDK includes a handful of new code analyzers that concern API compatibility, platform compatibility, trimming safety, use of span in string concatenation and splitting, faster string APIs, and faster collection APIs. For a full list of new (and removed) analyzers, see [Analyzer releases - .NET 6](#).

Custom platform guards

The [Platform compatibility analyzer](#) recognizes the `Is<Platform>` methods in the `OperatingSystem` class, for example, `OperatingSystem.IsWindows()`, as platform guards. To allow for custom platform guards, .NET 6 introduces two new attributes that you can use to annotate fields, properties, or methods with a supported or unsupported platform name:

- [SupportedOSPlatformGuardAttribute](#)
- [UnsupportedOSPlatformGuardAttribute](#)

Windows Forms

`Application.SetDefaultFont(Font)` is a new method in .NET 6 that sets the default font across your application.

The templates for C# Windows Forms apps have been updated to support `global using` directives, file-scoped namespaces, and nullable reference types. In addition, they include application bootstrap code, which reduces boilerplate code and allows the Windows Forms designer to render the design surface in the preferred font. The bootstrap code is a call to `ApplicationConfiguration.Initialize()`, which is a source-generated method that emits calls to other configuration methods, such as `Application.EnableVisualStyles()`. Additionally, if you set a non-default font via the `ApplicationDefaultFont` MSBuild property, `ApplicationConfiguration.Initialize()` emits a call to `SetFont(Font)`.

For more information, see the [What's new in Windows Forms](#) blog post.

Source build

The *source tarball*, which contains all the source for the .NET SDK, is now a product of the .NET SDK build. Other organizations, such as Red Hat, can build their own version of the SDK using this source tarball.

Target framework monikers

Additional OS-specific target framework monikers (TFMs) have been added for .NET 6, for example, `net6.0-android`, `net6.0-ios`, and `net6.0-macos`. For more information, see [.NET 5+ OS-specific TFMs](#).

Generic math

In preview is the ability to use operators on generic types in .NET 6. .NET 6 introduces numerous interfaces that make use of C# 10's new preview feature, `static abstract` interface members. These interfaces correspond to different operators, for example, `IAdditionOperators` represents the `+` operator. The interfaces are available in the `System.Runtime.Experimental` NuGet package. For more information, see the [Generic math](#) blog post.

NuGet package validation

If you're a NuGet library developer, new [package-validation tooling](#) enables you to validate that your packages are consistent and well-formed. You can determine if:

- There are any breaking changes across package versions.

- The package has the same set of public APIs for all runtime-specific implementations.
- There are any gaps for target-framework or runtime applicability.

For more information, see the [Package Validation](#) blog post.

Reflection APIs

.NET 6 introduces the following new APIs that inspect code and provide nullability information:

- [System.Reflection.NullabilityInfo](#)
- [System.Reflection.NullabilityInfoContext](#)
- [System.Reflection.NullabilityState](#)

These APIs are useful for reflection-based tools and serializers.

Microsoft.Extensions APIs

Several extensions namespaces have improvements in .NET 6, as the following table shows.

| Namespace | Improvements |
|--|--|
| Microsoft.Extensions.DependencyInjection | <code>CreateAsyncScope</code> lets you safely use a <code>using</code> statement for a service provider that registers an <code>IAsyncDisposable</code> service. |
| Microsoft.Extensions.Hosting | New <code>ConfigureHostOptions</code> methods simplify application setup. |
| Microsoft.Extensions.Logging | <code>Microsoft.Extensions.Logging</code> has a new source generator for performant logging APIs. The source generator is triggered if you add the new <code>LoggerMessageAttribute</code> to a <code>partial</code> logging method. At compile time, the generator generates the implementation of the <code>partial</code> method, which is typically faster at run time than existing logging solutions. For more information, see Compile-time logging source generation . |

New LINQ APIs

Numerous LINQ methods have been added in .NET 6. Most of the new methods listed in the following table have equivalent methods in the [System.Linq.Queryable](#) type.

| Method | Description |
|--|--|
| Enumerable.TryGetNonEnumeratedCount<TSource>(IEnumerable<TSource>, Int32) | Attempts to determine the number of elements in a sequence without forcing an enumeration. |
| Enumerable.Chunk<TSource>(IEnumerable<TSource>, Int32) | Splits the elements of a sequence into chunks of a specified size. |
| Enumerable.MaxBy and Enumerable.MinBy | Finds maximal or minimal elements using a key selector. |
| Enumerable.DistinctBy , Enumerable.ExceptBy , Enumerable.IntersectBy , and Enumerable.UnionBy | These new variations of methods that perform set-based operations let you specify equality using a key selector function. |
| Enumerable.ElementAt<TSource>(IEnumerable<TSource>, Index) and Enumerable.ElementAtOrDefault<TSource>(IEnumerable<TSource>, Index) | Accepts indexes counted from the beginning or end of the sequence—for example, <code>Enumerable.Range(1, 10).ElementAt(^2)</code> returns 9. |
| Enumerable.FirstOrDefault<TSource>(IEnumerable<TSource>, TSource) and Enumerable.FirstOrDefault<TSource>(IEnumerable<TSource>, Func<TSource, Boolean>, TSource) Enumerable.LastOrDefault<TSource>(IEnumerable<TSource>, TSource) and Enumerable.LastOrDefault<TSource>(IEnumerable<TSource>, Func<TSource, Boolean>, TSource) Enumerable.SingleOrDefault<TSource>(IEnumerable<TSource>, TSource) and Enumerable.SingleOrDefault<TSource>(IEnumerable<TSource>, Func<TSource, Boolean>, TSource) | New overloads let you specify a default value to use if the sequence is empty. |
| Enumerable.Max<TSource>(IEnumerable<TSource>, IComparer<TSource>) and Enumerable.Min<TSource>(IEnumerable<TSource>, IComparer<TSource>) | New overloads let you specify a comparer. |
| Enumerable.Take<TSource>(IEnumerable<TSource>, Range) | Accepts a Range argument to simplify taking a slice of a sequence—for example, you can use <code>source.Take(2..7)</code> instead of <code>source.Take(7).Skip(2)</code> . |

| Method | Description |
|---|--|
| <code>Enumerable.Zip<TFirst,TSecond,TThird></code> <code>(IEnumerable<TFirst>, IEnumerable<TSecond>,</code> <code>IEnumerable<TThird>)</code> | Produces a sequence of tuples with elements from <i>three</i> specified sequences. |

Date, time, and time zone improvements

The following two structs were added in .NET 6: [System.DateOnly](#) and [System.TimeOnly](#). These represent the date part and the time part of a [DateTime](#), respectively. [DateOnly](#) is useful for birthdays and anniversaries, and [TimeOnly](#) is useful for daily alarms and weekly business hours.

You can now use either Internet Assigned Numbers Authority (IANA) or Windows time zone IDs on any operating system that has time zone data installed. The [TimeZoneInfo.FindSystemTimeZoneByIld\(String\)](#) method has been updated to automatically convert its input from a Windows time zone to an IANA time zone (or vice versa) if the requested time zone is not found on the system. In addition, the new methods [TryConvertIanaIdToWindowsId\(String, String\)](#) and [TryConvertWindowsIdToIanaId](#) have been added for scenarios when you still need to manually convert from one time zone format to another.

There are a few other time zone improvements as well. For more information, see [Date, Time, and Time Zone Enhancements in .NET 6 ↗](#).

PriorityQueue class

The new [PriorityQueue<TElement,TPriority>](#) class represents a collection of items that have both a value and a priority. Items are dequeued in increasing priority order—that is, the item with the lowest priority value is dequeued first. This class implements a [min heap ↗](#) data structure.

See also

- [What's new in C# 10](#)
- [What's new in F# 6](#)
- [What's new in EF Core 6](#)
- [What's new in ASP.NET Core 6](#)
- [Release notes for .NET 6 ↗](#)
- [Release notes for Visual Studio 2022](#)
- [Blog: Announcing .NET 6 ↗](#)

- Blog: Try the new System.Text.Json source generator ↗

 **Collaborate with us on GitHub**

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

.NET is an open source project.
Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

Breaking changes in .NET 6

Article • 11/21/2023

If you're migrating an app to .NET 6, the breaking changes listed here might affect you. Changes are grouped by technology area, such as ASP.NET Core or Windows Forms.

This article indicates whether each breaking change is *binary compatible* or *source compatible*:

- **Binary compatible** - Existing binaries will load and execute successfully without recompilation, and the run-time behavior won't change.
- **Source compatible** - Source code will compile successfully without changes when targeting the new runtime or using the new SDK or component.

ASP.NET Core

Expand table

| Title | Binary compatible | Source compatible |
|--|-------------------|-------------------|
| ActionResult<T> sets StatusCode to 200 | ✓ | ✗ |
| AddDataAnnotationsValidation method made obsolete | ✓ | ✗ |
| Assemblies removed from Microsoft.AspNetCore.App shared framework | ✗ | ✓ |
| Blazor: Parameter name changed in RequestImageFileAsync method | ✓ | ✗ |
| Blazor: WebEventDescriptorEventArgsType property replaced | ✗ | ✗ |
| Blazor: Byte array interop | ✓ | ✗ |
| Changed MessagePack library in @microsoft/signalr-protocol-msgpack | ✗ | ✓ |
| ClientCertificate property doesn't trigger renegotiation for HttpSys | ✓ | ✗ |
| EndpointName metadata not set automatically | ✓ | ✗ |
| Identity: Default Bootstrap version of UI changed | ✗ | ✗ |
| Kestrel: Log message attributes changed | ✓ | ✗ |

| Title | Binary compatible | Source compatible |
|--|-------------------|-------------------|
| Microsoft.AspNetCore.Http.Features split | ✗ | ✓ |
| Middleware: HTTPS Redirection Middleware throws exception on ambiguous HTTPS ports | ✓ | ✗ |
| Middleware: New Use overload | ✓ | ✗ |
| Minimal API renames in RC 1 | ✗ | ✗ |
| Minimal API renames in RC 2 | ✗ | ✗ |
| MVC doesn't buffer IAsyncEnumerable types when using System.Text.Json | ✓ | ✗ |
| Nullable reference type annotations changed | ✓ | ✗ |
| Obsoleted and removed APIs | ✓ | ✗ |
| PreserveCompilationContext not configured by default | ✗ | ✓ |
| Razor: Compiler no longer produces a Views assembly | ✓ | ✗ |
| Razor: Logging ID changes | ✗ | ✓ |
| Razor: RazorEngine APIs marked obsolete | ✓ | ✗ |
| SignalR: Java Client updated to RxJava3 | ✗ | ✓ |
| TryParse and BindAsync methods are validated | ✗ | ✗ |

Containers

[+] Expand table

| Title | Binary compatible | Source compatible |
|---|-------------------|-------------------|
| Default console logger formatting in container images | ✓ | ✗ |

For information on other breaking changes for containers in .NET 6, see [.NET 6 Container Release Notes](#).

Core .NET libraries

[Expand table](#)

| Title | Binary compatible | Source compatible |
|--|-------------------|-------------------|
| API obsoletions with non-default diagnostic IDs | ✓ | ✗ |
| Changes to nullable reference type annotations | ✓ | ✗ |
| Conditional string evaluation in Debug methods | ✓ | ✗ |
| Environment.ProcessorCount behavior on Windows | ✓ | ✗ |
| EventSource callback behavior | ✓ | ✓ |
| File.Replace on Unix throws exceptions to match Windows | ✓ | ✗ |
| FileStream locks files with shared lock on Unix | ✗ | ✓ |
| FileStream no longer synchronizes file offset with OS | ✗ | ✗ |
| FileStream.Position updates after ReadAsync or WriteAsync completes | ✗ | ✗ |
| New diagnostic IDs for obsoleted APIs | ✓ | ✗ |
| New System.Linq.Queryable method overloads | ✓ | ✗ |
| Older framework versions dropped from package | ✗ | ✓ |
| Parameter names changed | ✓ | ✗ |
| Parameter names in Stream-derived types | ✓ | ✗ |
| Partial and zero-byte reads in DeflateStream, GZipStream, and CryptoStream | ✓ | ✗ |
| Set timestamp on read-only file on Windows | ✗ | ✓ |
| Standard numeric format parsing precision | ✓ | ✗ |
| Static abstract members in interfaces | ✗ | ✓ |
| StringBuilder.Append overloads and evaluation order | ✗ | ✓ |
| Strong-name APIs throw PlatformNotSupportedException | ✗ | ✓ |
| System.Drawing.Common only supported on Windows | ✗ | ✗ |
| System.Security.SecurityContext is marked obsolete | ✓ | ✗ |
| Task.FromResult may return singleton | ✗ | ✓ |

| Title | Binary compatible | Source compatible |
|---|-------------------|-------------------|
| Unhandled exceptions from a BackgroundService | ✓ | ✗ |

Cryptography

[Expand table](#)

| Title | Binary compatible | Source compatible |
|---|-------------------|-------------------|
| CreateEncryptor methods throw exception for incorrect feedback size | ✗ | ✓ |

Deployment

[Expand table](#)

| Title | Binary compatible | Source compatible |
|---------------------------------|-------------------|-------------------|
| x86 host path on 64-bit Windows | ✓ | ✓ |

Entity Framework Core

[Breaking changes in EF Core 6](#)

Extensions

[Expand table](#)

| Title | Binary compatible | Source compatible |
|--|-------------------|-------------------|
| AddProvider checks for non-null provider | ✓ | ✗ |
| FileConfigurationProvider.Load throws InvalidDataException | ✓ | ✗ |
| Repeated XML elements include index | ✗ | ✓ |
| Resolving disposed ServiceProvider throws exception | ✓ | ✗ |

Globalization

[\[+\] Expand table](#)

| Title | Binary compatible | Source compatible |
|---|-------------------|-------------------|
| Culture creation and case mapping in globalization-invariant mode | | |

Interop

[\[+\] Expand table](#)

| Title | Binary compatible | Source compatible |
|---------------------------------------|-------------------|-------------------|
| Static abstract members in interfaces | ✗ | ✓ |

JIT compiler

[\[+\] Expand table](#)

| Title | Binary compatible | Source compatible |
|---|-------------------|-------------------|
| Coerce call arguments according to ECMA-335 | ✓ | ✓ |

Networking

[\[+\] Expand table](#)

| Title | Binary compatible | Source compatible |
|--|-------------------|-------------------|
| Port removed from SPN for Kerberos and Negotiate | ✗ | ✓ |
| WebRequest, WebClient, and ServicePoint are obsolete | ✓ | ✗ |

SDK

[Expand table](#)

| Title | Binary compatible | Source compatible |
|---|-------------------|-------------------|
| -p option for dotnet run is deprecated | ✓ | ✗ |
| C# code in templates not supported by earlier versions | ✓ | ✓ |
| EditorConfig files implicitly included | ✓ | ✗ |
| Generate apphost for macOS | ✓ | ✗ |
| Generate error for duplicate files in publish output | ✗ | ✓ |
| GetTargetFrameworkProperties and GetNearestTargetFramework removed from ProjectReference protocol | ✗ | ✓ |
| Install location for x64 emulated on Arm64 | ✓ | ✗ |
| MSBuild no longer supports calling GetType() | | |
| .NET can't be installed to custom location | ✓ | ✓ |
| OutputType not automatically set to WinExe | ✓ | ✗ |
| Publish ReadyToRun with --no-restore requires changes | ✓ | ✗ |
| runtimeconfig.dev.json file not generated | ✗ | ✓ |
| Runtimeldentifier warning if self-contained is unspecified | ✓ | ✗ |
| Tool manifests in root folder | ✓ | ✓ |
| Version requirements for .NET 6 SDK | ✓ | ✓ |
| .version file includes build version | ✓ | ✓ |
| Write reference assemblies to IntermediateOutputPath | ✗ | ✓ |

Serialization

[Expand table](#)

| Title | Binary compatible | Source compatible |
|---|-------------------|-------------------|
| DataContractSerializer retains sign when deserializing -0 | ✗ | ✓ |

| Title | Binary compatible | Source compatible |
|--|-------------------|-------------------|
| Default serialization format for TimeSpan | ✗ | ✓ |
| IAsyncEnumerable serialization | ✓ | ✗ |
| JSON source-generation API refactoring | ✗ | ✓ |
| JsonNumberHandlingAttribute on collection properties | ✗ | ✓ |
| New JsonSerializer source generator overloads | ✗ | ✓ |

Windows Forms

[\[+\] Expand table](#)

| Title | Binary compatible | Source compatible |
|--|-------------------|-------------------|
| C# templates use application bootstrap | ✓ | ✗ |
| Selected TableLayoutSettings properties throw InvalidEnumArgumentException | ✗ | ✓ |
| DataGridView-related APIs now throw InvalidOperationException | ✗ | ✓ |
| ListViewGroupCollection methods throw new InvalidOperationException | ✗ | ✓ |
| NotifyIcon.Text maximum text length increased | ✗ | ✓ |
| ScaleControl called only when needed | ✓ | ✗ |
| Some APIs throw ArgumentNullException | ✗ | ✓ |
| TreeNodeCollection.Item throws exception if node is assigned elsewhere | ✗ | ✓ |

XML and XSLT

[\[+\] Expand table](#)

| Title | Binary compatible | Source compatible |
|---|-------------------|-------------------|
| XNodeReader.GetAttribute behavior for invalid index | ✓ | ✗ |

See also

- [What's new in .NET 6](#)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



.NET feedback

.NET is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

What's new in .NET 5

Article • 09/22/2022

.NET 5 is the next major release of .NET Core following 3.1. We named this new release .NET 5 instead of .NET Core 4 for two reasons:

- We skipped version numbers 4.x to avoid confusion with .NET Framework 4.x.
- We dropped "Core" from the name to emphasize that this is the main implementation of .NET going forward. .NET 5 supports more types of apps and more platforms than .NET Core or .NET Framework.

ASP.NET Core 5.0 is based on .NET 5 but retains the name "Core" to avoid confusing it with ASP.NET MVC 5. Likewise, Entity Framework Core 5.0 retains the name "Core" to avoid confusing it with Entity Framework 5 and 6.

.NET 5 includes the following improvements and new features compared to .NET Core 3.1:

- [C# updates](#)
- [F# updates](#)
- [Visual Basic updates](#)
- [System.Text.Json new features](#)
- [Single file apps](#)
- [App trimming ↗](#)
- Windows Arm64 and Arm64 intrinsics
- Tooling support for dump debugging
- The runtime libraries are 80% annotated for [nullable reference types](#)
- Performance improvements:
 - [Garbage Collection \(GC\) ↗](#)
 - [System.Text.Json ↗](#)
 - [System.Text.RegularExpressions ↗](#)
 - [Async ValueTask pooling ↗](#)
 - [Container size optimizations ↗](#)
 - [Many more areas ↗](#)

.NET 5 doesn't replace .NET Framework

.NET 5 and later versions are the main implementation of .NET going forward, but .NET Framework 4.x is still supported. There are no plans to port the following technologies from .NET Framework to .NET 5, but there are alternatives in .NET:

| Technology | Recommended alternative |
|-----------------------|--|
| Web Forms | ASP.NET Core Blazor or Razor Pages |
| Windows Workflow (WF) | Elsa-Workflows ↗ |

Windows Communication Foundation

The original implementation of [Windows Communication Foundation \(WCF\)](#) was only supported on Windows. However, there's a client port available from the .NET Foundation. It's entirely [open source](#) ↗, cross platform, and supported by Microsoft. The core NuGet packages are listed below:

- [System.ServiceModel.Duplex](#) ↗
- [System.ServiceModel.Federation](#) ↗
- [System.ServiceModel.Http](#) ↗
- [System.ServiceModel.NetTcp](#) ↗
- [System.ServiceModel.Primitives](#) ↗
- [System.ServiceModel.Security](#) ↗

The server components that complement the aforementioned client libraries are available through [CoreWCF](#) ↗. As of April 2022, CoreWCF is officially supported by Microsoft. However, for an alternative to WCF, consider [gRPC](#).

.NET 5 doesn't replace .NET Standard

New application development can specify the `net5.0` Target Framework Moniker (TFM) for all project types, including class libraries. Sharing code between .NET 5 workloads is simplified: all you need is the `net5.0` TFM.

For .NET 5 apps and libraries, the `net5.0` TFM combines and replaces the `netcoreapp` and `netstandard` TFMs. However, if you plan to share code between .NET Framework, .NET Core, and .NET 5 workloads, you can do so by specifying `netstandard2.0` as your TFM. For more information, see [.NET Standard](#).

C# updates

Developers writing .NET 5 apps will have access to the latest C# version and features. .NET 5 is paired with C# 9, which brings many new features to the language. Here are a few highlights:

- **Records:** Reference types with value-based equality semantics and non-destructive mutation supported by a new `with` expression.
- **Relational pattern matching:** Extends pattern matching capabilities to relational operators for comparative evaluations and expressions, including logical patterns - new keywords `and`, `or`, and `not`.
- **Top-level statements:** As a means for accelerating the adoption and learning of C#, the `Main` method can be omitted, and an application as simple as the following example is valid:

C#

```
System.Console.WriteLine("Hello world!");
```

- **Function pointers:** Language constructs that expose the following intermediate language (IL) opcodes: `ldftn` and `calli`.

For more information on the available C# 9 features, see [What's new in C# 9](#).

Source generators

In addition to some of the highlighted new C# features, source generators are making their way into developer projects. Source generators allow code that runs during compilation to inspect your program and produce additional files that are compiled together with the rest of your code.

For more information on source generators, see [Introducing C# source generators](#) and [C# source generator samples](#).

F# updates

F# is the .NET functional programming language, and with .NET 5, developers have access to F# 5. One of the new features is interpolated strings, similar to interpolated strings in C#, and even JavaScript.

F#

```
let name = "David"
let age = 36
let message = $"{name} is {age} years old."
```

In addition to basic string interpolation, there's typed interpolation. With typed interpolation, a given type must match the format specifier.

```
F#  
  
let name = "David"  
let age = 36  
let message = $"{name} is {age} years old."
```

This format is similar to the [sprintf](#) function that formats a string based on type-safe inputs.

For more information, see [What's new in F# 5](#).

Visual Basic updates

There are no new language features for Visual Basic in .NET 5. However, with .NET 5, Visual Basic support is extended to:

| Description | dotnet new parameter |
|--|----------------------|
| Console Application | console |
| Class library | classlib |
| WPF Application | wpf |
| WPF Class library | wpflib |
| WPF Custom Control Library | wpfcustomcontrollib |
| WPF User Control Library | wpfusercontrollib |
| Windows Forms (WinForms) Application | winforms |
| Windows Forms (WinForms) Class library | winformslib |
| Unit Test Project | mstest |
| NUnit 3 Test Project | nunit |
| NUnit 3 Test Item | nunit-test |
| xUnit Test Project | xunit |

For more information on project templates from the .NET CLI, see [dotnet new](#).

System.Text.Json new features

There are new features in and for [System.Text.Json](#):

- Preserve references and handle circular references
- HttpClient and HttpContent extension methods
- Allow or write numbers in quotes
- Support immutable types and C# 9 Records
- Support non-public property accessors
- Support fields
- Conditionally ignore properties
- Support non-string-key dictionaries
- Allow custom converters to handle null
- Copy JsonSerializerOptions
- Create JsonSerializerOptions with web defaults

See also

- [The Journey to one .NET](#)
- [Performance improvements in .NET 5 ↗](#)
- [Download the .NET SDK ↗](#)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

Breaking changes in .NET 5

Article • 03/18/2023

If you're migrating an app to .NET 5, the breaking changes listed here might affect you. Changes are grouped by technology area, such as ASP.NET Core or cryptography.

This article indicates whether each breaking change is *binary compatible* or *source compatible*:

- **Binary compatible** - Existing binaries will load and execute successfully without recompilation, and the run-time behavior won't change.
- **Source compatible** - Source code will compile successfully without changes when targeting the new runtime or using the new SDK or component.

ASP.NET Core

| Title | Binary compatible | Source compatible |
|--|-------------------|-------------------|
| ASP.NET Core apps deserialize quoted numbers | ✓ | ✗ |
| AzureAD.UI and AzureADB2C.UI APIs obsolete | ✓ | ✗ |
| BinaryFormatter serialization methods are obsolete | ✓ | ✗ |
| Resource in endpoint routing is HttpContext | ✓ | ✗ |
| Microsoft-prefixed Azure integration packages removed | ✗ | ✓ |
| Blazor: Route precedence logic changed in Blazor apps | ✓ | ✗ |
| Blazor: Updated browser support | ✓ | ✓ |
| Blazor: Insignificant whitespace trimmed by compiler | ✓ | ✗ |
| Blazor: JObjectReference and JSInProcessObjectReference types are internal | ✓ | ✗ |
| Blazor: Target framework of NuGet packages changed | ✗ | ✓ |
| Blazor: ProtectedBrowserStorage feature moved to shared framework | ✓ | ✗ |
| Blazor: RenderTreeFrame readonly public fields are now properties | ✗ | ✓ |
| Blazor: Updated validation logic for static web assets | ✗ | ✓ |

| Title | Binary compatible | Source compatible |
|--|-------------------|-------------------|
| Cryptography APIs not supported on browser | ✗ | ✓ |
| Extensions: Package reference changes | ✗ | ✓ |
| Kestrel and IIS BadHttpRequestException types are obsolete | ✓ | ✗ |
| HttpClient instances created by IHttpClientFactory log integer status codes | ✓ | ✗ |
| HttpSys: Client certificate renegotiation disabled by default | ✓ | ✗ |
| IIS: UrlRewrite middleware query strings are preserved | ✓ | ✗ |
| Kestrel: Configuration changes detected by default | ✓ | ✗ |
| Kestrel: Default supported TLS protocol versions changed | ✓ | ✗ |
| Kestrel: HTTP/2 disabled over TLS on incompatible Windows versions | ✓ | ✓ |
| Kestrel: Libuv transport marked as obsolete | ✓ | ✗ |
| Obsolete properties on ConsoleLoggerOptions | ✓ | ✗ |
| ResourceManagerWithCultureStringLocalizer class and WithCulture interface member removed | ✓ | ✗ |
| Pubternal APIs removed | ✓ | ✗ |
| Obsolete constructor removed in request localization middleware | ✓ | ✗ |
| Middleware: Database error page marked as obsolete | ✓ | ✗ |
| Exception handler middleware throws original exception | ✓ | ✓ |
| ObjectModelValidator calls a new overload of Validate | ✓ | ✗ |
| Cookie name encoding removed | ✓ | ✗ |
| IdentityModel NuGet package versions updated | ✗ | ✓ |
| SignalR: MessagePack Hub Protocol options type changed | ✓ | ✗ |
| SignalR: MessagePack Hub Protocol moved | ✓ | ✗ |
| UseSignalR and UseConnections methods removed | ✓ | ✗ |
| CSV content type changed to standards-compliant | ✓ | ✗ |

Code analysis

| Title | Binary compatible | Source compatible |
|----------------|-------------------|-------------------|
| CA1416 warning | ✓ | ✗ |
| CA1417 warning | ✓ | ✗ |
| CA1831 warning | ✓ | ✗ |
| CA2013 warning | ✓ | ✗ |
| CA2014 warning | ✓ | ✗ |
| CA2015 warning | ✓ | ✗ |
| CA2200 warning | ✓ | ✗ |
| CA2247 warning | ✓ | ✗ |

Core .NET libraries

| Title | Binary compatible | Source compatible |
|--|-------------------|-------------------|
| Assembly-related API changes for single-file publishing | ✗ | ✓ |
| BinaryFormatter serialization methods are obsolete | ✓ | ✗ |
| Code access security APIs are obsolete | ✓ | ✗ |
| CreateCounterSetInstance throws InvalidOperationException | ✓ | ✗ |
| Default ActivityIdFormat is W3C | ✗ | ✓ |
| Environment.OSVersion returns the correct version | ✗ | ✓ |
| FrameworkDescription's value is .NET not .NET Core | ✓ | ✗ |
| GAC APIs are obsolete | ✓ | ✗ |
| Hardware intrinsic IsSupported checks | ✗ | ✓ |
| IntPtr and UIntPtr implement IFormattable | ✓ | ✗ |
| LastIndexOf handles empty search strings | ✗ | ✓ |
| URI paths with non-ASCII characters on Unix | ✗ | ✓ |

| Title | Binary compatible | Source compatible |
|---|-------------------|-------------------|
| API obsolesions with non-default diagnostic IDs | ✓ | ✗ |
| Obsolete properties on ConsoleLoggerOptions | ✓ | ✗ |
| Complexity of LINQ OrderBy.First | ✗ | ✓ |
| OSPlatform attributes renamed or removed | ✓ | ✗ |
| Microsoft.DotNet.PlatformAbstractions package removed | ✗ | ✓ |
| PrincipalPermissionAttribute is obsolete | ✓ | ✗ |
| Parameter name changes from preview versions | ✓ | ✗ |
| Parameter name changes in reference assemblies | ✓ | ✗ |
| Remoting APIs are obsolete | ✗ | ✓ |
| Order of Activity.Tags list is reversed | ✓ | ✗ |
| SSE and SSE2 comparison methods handle NaN | ✓ | ✗ |
| Thread.Abort is obsolete | ✓ | ✗ |
| Uri recognition of UNC paths on Unix | ✗ | ✓ |
| UTF-7 code paths are obsolete | ✓ | ✗ |
| Behavior change for Vector2.Lerp and Vector4.Lerp | ✓ | ✗ |
| Vector<T> throws NotSupportedException | ✗ | ✓ |

Cryptography

| Title | Binary compatible | Source compatible |
|---|-------------------|-------------------|
| Cryptography APIs not supported on browser | ✗ | ✓ |
| Cryptography.Oid is init-only | ✓ | ✗ |
| Default TLS cipher suites on Linux | ✗ | ✓ |
| Create() overloads on cryptographic abstractions are obsolete | ✓ | ✗ |
| Default FeedbackSize value changed | ✓ | ✗ |

Entity Framework Core

Breaking changes in EF Core 5.0

Globalization

| Title | Binary compatible | Source compatible |
|--|-------------------|-------------------|
| Use ICU libraries on Windows | ✗ | ✓ |
| StringInfo and TextElementEnumerator are UAX29-compliant | ✗ | ✓ |
| Unicode category changed for Latin-1 characters | ✓ | ✗ |
| TextInfo.ListSeparator values changed | ✓ | ✗ |

Interop

| Title | Binary compatible | Source compatible |
|---|-------------------|-------------------|
| Support for WinRT is removed | ✗ | ✓ |
| Casting RCW to InterfacesInspectable throws exception | ✗ | ✓ |
| No A/W suffix probing on non-Windows platforms | ✗ | ✓ |

Networking

| Title | Binary compatible | Source compatible |
|--|-------------------|-------------------|
| Cookie path handling conforms to RFC 6265 | ✓ | ✗ |
| LocalEndPoint is updated after calling SendToAsync | ✓ | ✗ |
| MulticastOption.Group doesn't accept null | ✓ | ✗ |
| Streams allow successive Begin operations | ✗ | ✓ |
| WinHttpHandler removed from .NET runtime | ✗ | ✓ |

SDK

| Title | Binary compatible | Source compatible |
|---|-------------------|-------------------|
| Directory.Packages.props files imported by default | ✗ | ✓ |
| Error generated when executable project references mismatched executable | | ✓ |
| FrameworkReference replaced with WindowsSdkPackageVersion for Windows SDK | ✓ | ✗ |
| NETCOREAPP3_1 preprocessor symbol not defined | ✓ | ✗ |
| OutputType set to WinExe | ✗ | ✓ |
| PublishDepsFilePath behavior change | ✗ | ✓ |
| TargetFramework change from netcoreapp to net | ✗ | ✓ |
| WinForms and WPF apps use Microsoft.NET.Sdk | ✗ | ✓ |

Security

| Title | Binary compatible | Source compatible |
|--|-------------------|-------------------|
| Code access security APIs are obsolete | ✓ | ✗ |
| PrincipalPermissionAttribute is obsolete | ✓ | ✗ |
| UTF-7 code paths are obsolete | ✓ | ✗ |

Serialization

| Title | Binary compatible | Source compatible |
|---|-------------------|-------------------|
| BinaryFormatter.Deserialize rewraps exceptions | ✓ | ✗ |
| JsonSerializer.Deserialize requires single-character string | ✓ | ✗ |
| ASP.NET Core apps deserialize quoted numbers | ✓ | ✗ |
| JsonSerializer.Serialize throws ArgumentNullException | ✓ | ✗ |
| Non-public, parameterless constructors not used for | ✓ | ✗ |

| Title | Binary compatible | Source compatible |
|--|-------------------|-------------------|
| deserialization | | |
| Options are honored when serializing key-value pairs | ✓ | ✗ |

Windows Forms

| Title | Binary compatible | Source compatible |
|---|-------------------|-------------------|
| Native code can't access Windows Forms objects | ✓ | ✗ |
| OutputType set to WinExe | ✗ | ✓ |
| DataGridView doesn't reset custom fonts | ✓ | ✗ |
| Methods throw ArgumentException | ✓ | ✗ |
| Methods throw ArgumentNullException | ✓ | ✗ |
| Properties throw ArgumentOutOfRangeException | ✓ | ✗ |
| TextFormatFlags.ModifyString is obsolete | ✓ | ✗ |
| DataGridView APIs throw InvalidOperationException | ✓ | ✗ |
| WinForms apps use Microsoft.NET.Sdk | ✗ | ✓ |
| Removed status bar controls | ✓ | ✗ |

WPF

| Title | Binary compatible | Source compatible |
|--------------------------------|-------------------|-------------------|
| OutputType set to WinExe | ✗ | ✓ |
| WPF apps use Microsoft.NET.Sdk | ✗ | ✓ |

See also

- [What's new in .NET 5](#)

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

What's new in .NET Core 3.1

Article • 07/12/2022

This article describes what is new in .NET Core 3.1. This release contains minor improvements to .NET Core 3.0, focusing on small, but important, fixes. The most important feature about .NET Core 3.1 is that it's a [long-term support \(LTS\)](#) release.

If you're using Visual Studio 2019, you must update to [Visual Studio 2019 version 16.4 or later](#) to work with .NET Core 3.1 projects. For information on what's new in Visual Studio version 16.4, see [What's New in Visual Studio 2019 version 16.4](#).

Visual Studio for Mac also supports and includes .NET Core 3.1 in Visual Studio for Mac 8.4.

For more information about the release, see the [.NET Core 3.1 announcement](#).

- [Download and get started with .NET Core 3.1](#) on Windows, macOS, or Linux.

Long-term support

.NET Core 3.1 is an LTS release with support from Microsoft for three years after its release. It's highly recommended that you move your apps to the latest LTS release. See the [.NET and .NET Core support policy](#) page for a list of supported releases.

| Release | End of life date |
|---------------|-----------------------------------|
| .NET Core 3.1 | End of life on December 13, 2022. |
| .NET Core 3.0 | End of life on March 3, 2020. |
| .NET Core 2.2 | End of life on December 23, 2019. |
| .NET Core 2.1 | End of life on August 21, 2021. |

For more information, see the [.NET and .NET Core support policy](#).

macOS appHost and notarization

macOS only

Starting with the notarized .NET Core SDK 3.1 for macOS, the appHost setting is disabled by default. For more information, see [macOS Catalina Notarization and the impact on .NET Core downloads and projects](#).

When the `appHost` setting is enabled, .NET Core generates a native Mach-O executable when you build or publish. Your app runs in the context of the `appHost` when it is run from source code with the `dotnet run` command, or by starting the Mach-O executable directly.

Without the `appHost`, the only way a user can start a [framework-dependent](#) app is with the `dotnet <filename.dll>` command. An `appHost` is always created when you publish your app [self-contained](#).

You can either configure the `appHost` at the project level, or toggle the `appHost` for a specific `dotnet` command with the `-p:UseAppHost` parameter:

- Project file

XML

```
<PropertyGroup>
  <UseAppHost>true</UseAppHost>
</PropertyGroup>
```

- Command-line parameter

.NET CLI

```
dotnet run -p:UseAppHost=true
```

For more information about the `UseAppHost` setting, see [MSBuild properties for Microsoft.NET.Sdk](#).

Windows Forms

Windows only

 **Warning**

There are breaking changes in Windows Forms.

Legacy controls were included in Windows Forms that have been unavailable in the Visual Studio Designer Toolbox for some time. These were replaced with new controls back in .NET Framework 2.0. These have been removed from the Desktop SDK for .NET Core 3.1.

| Removed control | Recommended replacement | Associated APIs removed |
|-----------------|--|--|
| DataGrid | DataGridView | DataGridCell DataGridRow DataGridTableCollection DataGridColumnCollection DataGridTableStyle DataGridColumnStyle DataGridLineStyle DataGridParentRowsLabel DataGridParentRowsLabelStyle DataGridBoolColumn DataGridTextBox GridColumnStylesCollection GridTableStylesCollection HitTestType |
| ToolBar | ToolStrip | ToolBarAppearance |
| ToolBarButton | ToolStripButton | ToolBarButtonClickEventArgs ToolBarButtonClickEventHandler ToolBarButtonStyle ToolBarTextAlign |
| ContextMenu | ContextMenuStrip | |
| Menu | ToolStripDropDown ToolStripDropDownMenu | MenuItemCollection |
| MainMenu | MenuStrip | |
| MenuItem | ToolStripMenuItem | |

We recommend you update your applications to .NET Core 3.1 and move to the replacement controls. Replacing the controls is a straightforward process, essentially "find and replace" on the type.

C++/CLI

Windows only

Support has been added for creating C++/CLI (also known as "managed C++") projects. Binaries produced from these projects are compatible with .NET Core 3.0 and later versions.

To add support for C++/CLI in Visual Studio 2019 version 16.4, install the [Desktop development with C++ workload](#). This workload adds two templates to Visual Studio:

- CLR Class Library (.NET Core)
- CLR Empty Project (.NET Core)

Next steps

- Review the breaking changes between .NET Core 3.0 and 3.1.
- Review the breaking changes in .NET Core 3.1 for Windows Forms apps.

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

Breaking changes in .NET Core 3.1

Article • 07/28/2023

If you're migrating to version 3.1 of .NET Core or ASP.NET Core, the breaking changes listed in this article may affect your app.

ASP.NET Core

HTTP: Browser SameSite changes impact authentication

Some browsers, such as Chrome and Firefox, made breaking changes to their implementations of `SameSite` for cookies. The changes impact remote authentication scenarios, such as OpenID Connect and WS-Federation, which must opt out by sending `SameSite=None`. However, `SameSite=None` breaks on iOS 12 and some older versions of other browsers. The app needs to sniff these versions and omit `SameSite`.

For discussion on this issue, see [dotnet/aspnetcore#14996](#).

Version introduced

3.1 Preview 1

Old behavior

`SameSite` is a 2016 draft standard extension to HTTP cookies. It's intended to mitigate Cross-Site Request Forgery (CSRF). This was originally designed as a feature the servers would opt into by adding the new parameters. ASP.NET Core 2.0 added initial support for `SameSite`.

New behavior

Google proposed a new draft standard that isn't backwards compatible. The standard changes the default mode to `Lax` and adds a new entry `None` to opt out. `Lax` suffices for most app cookies; however, it breaks cross-site scenarios like OpenID Connect and WS-Federation login. Most OAuth logins aren't affected because of differences in how the request flows. The new `None` parameter causes compatibility problems with clients that implemented the prior draft standard (for example, iOS 12). Chrome 80 will include the changes. See [SameSite Updates](#) for the Chrome product launch timeline.

ASP.NET Core 3.1 has been updated to implement the new `SameSite` behavior. The update redefines the behavior of `SameSiteMode.None` to emit `SameSite=None` and adds a new value `SameSiteMode.Unspecified` to omit the `SameSite` attribute. All cookie APIs now default to `Unspecified`, though some components that use cookies set values more specific to their scenarios such as the OpenID Connect correlation and nonce cookies.

For other recent changes in this area, see [HTTP: Some cookie SameSite defaults changed to None](#). In ASP.NET Core 3.0, most defaults were changed from `SameSiteMode.Lax` to `SameSiteMode.None` (but still using the prior standard).

Reason for change

Browser and specification changes as outlined in the preceding text.

Recommended action

Apps that interact with remote sites, such as through third-party login, need to:

- Test those scenarios on multiple browsers.
- Apply the cookie policy browser sniffing mitigation discussed in [Support older browsers](#).

For testing and browser sniffing instructions, see the following section.

Determine if you're affected

Test your web app using a client version that can opt into the new behavior. Chrome, Firefox, and Microsoft Edge Chromium all have new opt-in feature flags that can be used for testing. Verify that your app is compatible with older client versions after you've applied the patches, especially Safari. For more information, see [Support older browsers](#).

Chrome

Chrome 78 and later yield misleading test results. Those versions have a temporary mitigation in place and allow cookies less than two minutes old. With the appropriate test flags enabled, Chrome 76 and 77 yield more accurate results. To test the new behavior, toggle `chrome://flags/#same-site-by-default-cookies` to enabled. Chrome 75 and earlier are reported to fail with the new `None` setting. For more information, see [Support older browsers](#).

Google doesn't make older Chrome versions available. You can, however, download older versions of Chromium, which will suffice for testing. Follow the instructions at [Download Chromium ↗](#).

- [Chromium 76 Win64 ↗](#)
- [Chromium 74 Win64 ↗](#)

Safari

Safari 12 strictly implemented the prior draft and fails if it sees the new `None` value in cookies. This must be avoided via the browser sniffing code shown in [Support older browsers](#). Ensure you test Safari 12 and 13 as well as WebKit-based, OS-style logins using Microsoft Authentication Library (MSAL), Active Directory Authentication Library (ADAL), or whichever library you're using. The problem is dependent on the underlying OS version. OSX Mojave 10.14 and iOS 12 are known to have compatibility problems with the new behavior. Upgrading to OSX Catalina 10.15 or iOS 13 fixes the problem. Safari doesn't currently have an opt-in flag for testing the new specification behavior.

Firefox

Firefox support for the new standard can be tested on version 68 and later by opting in on the `about:config` page with the feature flag `network.cookie.sameSite.laxByDefault`. No compatibility issues have been reported on older versions of Firefox.

Microsoft Edge

While Microsoft Edge supports the old `SameSite` standard, as of version 44 it didn't have any compatibility problems with the new standard.

Microsoft Edge Chromium

The feature flag is `edge://flags/#same-site-by-default-cookies`. No compatibility issues were observed when testing with Microsoft Edge Chromium 78.

Electron

Versions of Electron include older versions of Chromium. For example, the version of Electron used by Microsoft Teams is Chromium 66, which exhibits the older behavior. Perform your own compatibility testing with the version of Electron your product uses. For more information, see [Support older browsers](#).

Support older browsers

The 2016 `SameSite` standard mandated that unknown values be treated as `SameSite=Strict` values. Consequently, any older browsers that support the original standard may break when they see a `SameSite` property with a value of `None`. Web apps must implement browser sniffing if they intend to support these old browsers. ASP.NET Core doesn't implement browser sniffing for you because `User-Agent` request header values are highly unstable and change on a weekly basis. Instead, an extension point in the cookie policy allows you to add `User-Agent`-specific logic.

In `Startup.cs`, add the following code:

C#

```
private void CheckSameSite(HttpContext httpContext, CookieOptions options)
{
    if (options.SameSite == SameSiteMode.None)
    {
        var userAgent = httpContext.Request.Headers["User-Agent"].ToString();
        // TODO: Use your User Agent library of choice here.
        if /* UserAgent doesn't support new behavior */
        {
            options.SameSite = SameSiteMode.Unspecified;
        }
    }
}

public void ConfigureServices(IServiceCollection services)
{
    services.Configure<CookiePolicyOptions>(options =>
    {
        options.MinimumSameSitePolicy = SameSiteMode.Unspecified;
        options.OnAppendCookie = cookieContext =>
            CheckSameSite(cookieContext.Context,
cookieContext.CookieOptions);
        options.OnDeleteCookie = cookieContext =>
            CheckSameSite(cookieContext.Context,
cookieContext.CookieOptions);
    });
}

public void Configure(IApplicationBuilder app)
{
    // Before UseAuthentication or anything else that writes cookies.
    app.UseCookiePolicy();

    app.UseAuthentication();
    // code omitted for brevity
}
```

Opt-out switches

The `Microsoft.AspNetCore.SuppressSameSiteNone` compatibility switch enables you to temporarily opt out of the new ASP.NET Core cookie behavior. Add the following JSON to a `runtimeconfig.template.json` file in your project:

JSON

```
{  
  "configProperties": {  
    "Microsoft.AspNetCore.SuppressSameSiteNone": "true"  
  }  
}
```

Other Versions

Related `SameSite` patches are forthcoming for:

- ASP.NET Core 2.1, 2.2, and 3.0
- `Microsoft.Owin` 4.1
- `System.Web` (for .NET Framework 4.7.2 and later)

Category

ASP.NET

Affected APIs

- `Microsoft.AspNetCore.Builder.CookiePolicyOptions.MinimumSameSitePolicy`
- `Microsoft.AspNetCore.Http.CookieBuilder.SameSite`
- `Microsoft.AspNetCore.Http.CookieOptions.SameSite`
- `Microsoft.AspNetCore.Http.SameSiteMode`
- `Microsoft.Net.Http.Headers.SameSiteMode`
- `Microsoft.Net.Http.Headers.SetCookieHeaderValue.SameSite`

Deployment

[x86 host path on 64-bit Windows](#)

MSBuild

Design-time builds only return top-level package references

Starting in .NET Core SDK 3.1.400, only top-level package references are returned by the `RunResolvePackageDependencies` target.

Version introduced

.NET Core SDK 3.1.400

Change description

In previous versions of the .NET Core SDK, the `RunResolvePackageDependencies` target created the following MSBuild items that contained information from the NuGet assets file:

- `PackageDefinitions`
- `PackageDependencies`
- `TargetDefinitions`
- `FileDefinitions`
- `FileDependencies`

This data is used by Visual Studio to populate the Dependencies node in Solution Explorer. However, it can be a large amount of data, and the data isn't needed unless the Dependencies node is expanded.

Starting in the .NET Core SDK version 3.1.400, most of these items aren't generated by default. Only items of type `Package` are returned. If Visual Studio needs the items to populate the Dependencies node, it reads the information directly from the assets file.

Reason for change

This change was introduced to improve solution-load performance inside of Visual Studio. Previously, all package references would be loaded, which involved loading many references that most users would never view.

Recommended action

If you have MSBuild logic that depends on these items being created, set the `EmitLegacyAssetsFileItems` property to `true` in your project file. This setting enables the previous behavior where all the items are created.

Category

MSBuild

Affected APIs

N/A

SDK

[Tool manifests in root folder](#)

Windows Forms

Removed controls

Starting in .NET Core 3.1, some Windows Forms controls are no longer available.

Change description

Starting with .NET Core 3.1, various Windows Forms controls are no longer available. Replacement controls that have better design and support were introduced in .NET Framework 2.0. The deprecated controls were previously removed from designer toolboxes but were still available to be used.

The following types are no longer available:

- [ContextMenu](#)
- [DataGridView](#)
- [DataGridView.HitTestType](#)
- [DataGridViewBoolColumn](#)
- [DataGridViewCell](#)
- [DataGridViewCellStyle](#)
- [DataGridViewLineStyle](#)
- [DataGridViewParentRowsLabelStyle](#)
- [DataGridViewPreferredColumnWidthTypeConverter](#)
- [DataGridViewTableStyle](#)
- [DataGridViewTextBox](#)
- [DataGridViewTextBoxColumn](#)
- [GridColumnStylesCollection](#)

- [GridTablesFactory](#)
- [GridTableStylesCollection](#)
- [IDataGridEditingService](#)
- [IMenuEditorService](#)
- [MainMenu](#)
- [Menu](#)
- [Menu.MenuItemCollection](#)
- [MenuItem](#)
- [ToolBar](#)
- [ToolBarAppearance](#)
- [ToolBarButton](#)
- [ToolBar.ToolBarButtonCollection](#)
- [ToolBarButtonClickEventArgs](#)
- [ToolBarButtonStyle](#)
- [ToolBar.TextAlign](#)

Version introduced

3.1

Recommended action

Each removed control has a recommended replacement control. Refer to the following table:

| Removed control (API) | Recommended replacement | Associated APIs that are removed |
|------------------------------|--|--|
| ContextMenu | ContextMenuStrip | |
| DataGrid | DataGridView | DataGridCell, DataGridRow, DataGridTableCollection, DataGridColumnCollection, DataGridTableStyle, DataGridColumnStyle, DataGridLineStyle, DataGridParentRowsLabel, DataGridParentRowsLabelStyle, DataGridBoolColumn, DataGridTextBox, GridColumnStylesCollection, GridTableStylesCollection, HitTestType |
| MainMenu | MenuStrip | |
| Menu | ToolStripDropDown, ToolStripDropDownMenu | MenuItemCollection |

| Removed control (API) | Recommended replacement | Associated APIs that are removed |
|------------------------------|--------------------------------|---|
| MenuItem | ToolStripMenuItem | |
| ToolBar | ToolStrip | ToolBarAppearance |
| ToolBarButton | ToolStripButton | ToolBarButtonClickEventArgs, ToolBarButtonClickEventHandler, ToolBarButtonStyle, ToolBarTextAlign |

Category

Windows Forms

Affected APIs

- [System.Windows.Forms.ContextMenu](#)
- [System.Windows.Forms.GridColumnStylesCollection](#)
- [System.Windows.Forms.GridTablesFactory](#)
- [System.Windows.Forms.GridColumnStylesCollection](#)
- [System.Windows.Forms.IDataGridEditingService](#)
- [System.Windows.Forms.MainMenu](#)
- [System.Windows.Forms.Menu](#)
- [System.Windows.Forms.Menu.MenuItemCollection](#)
- [System.Windows.Forms.MenuItem](#)
- [System.Windows.Forms.ToolBar](#)
- [System.Windows.Forms.ToolBar.ToolBarButtonCollection](#)
- [System.Windows.Forms.ToolBarAppearance](#)
- [System.Windows.Forms.ToolBarButton](#)
- [System.Windows.Forms.ToolBarButtonClickEventArgs](#)
- [System.Windows.Forms.ToolBarButtonStyle](#)
- [System.Windows.Forms.ToolBarTextAlign](#)
- [System.Windows.Forms.DataGrid](#)
- [System.Windows.Forms.DataGrid.HitTestType](#)
- [System.Windows.Forms.DataGridBoolColumn](#)
- [System.Windows.Forms.DataGridCell](#)
- [System.Windows.Forms.DataGridViewColumnStyle](#)
- [System.Windows.Forms.DataGridViewLineStyle](#)
- [System.Windows.Forms.DataGridViewParentRowsLabelStyle](#)
- [System.Windows.Forms.DataGridViewColumnWidthTypeConverter](#)
- [System.Windows.Forms.DataGridViewColumnStyle](#)

- [System.Windows.Forms.DataGridTextBox](#)
 - [System.Windows.Forms.DataGridTextBoxColumn](#)
 - [System.Windows.Forms.Design.IMenuEditorService](#)
-

CellFormatting event not raised if tooltip is shown

A [DataGridView](#) now shows a cell's text and error tooltips when hovered by a mouse and when selected via the keyboard. If a tooltip is shown, the [DataGridView.CellFormatting](#) event is not raised.

Change description

Prior to .NET Core 3.1, a [DataGridView](#) that had the [ShowCellToolTips](#) property set to `true` showed a tooltip for a cell's text and errors when the cell was hovered by a mouse. Tooltips were not shown when a cell was selected via the keyboard (for example, by using the Tab key, shortcut keys, or arrow navigation). If the user edited a cell, and then, while the [DataGridView](#) was still in edit mode, hovered over a cell that did not have the [ToolTipText](#) property set, a [CellFormatting](#) event was raised to format the cell's text for display in the cell.

To meet accessibility standards, starting in .NET Core 3.1, a [DataGridView](#) that has the [ShowCellToolTips](#) property set to `true` shows tooltips for a cell's text and errors not only when the cell is hovered, but also when it's selected via the keyboard. As a consequence of this change, the [CellFormatting](#) event is *not* raised when cells that don't have the [ToolTipText](#) property set are hovered while the [DataGridView](#) is in edit mode. The event is not raised because the content of the hovered cell is shown as a tooltip instead of being displayed in the cell.

Version introduced

3.1

Recommended action

Refactor any code that depends on the [CellFormatting](#) event while the [DataGridView](#) is in edit mode.

Category

Windows Forms

Affected APIs

None

See also

- [What's new in .NET Core 3.1](#)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

What's new in .NET Core 3.0

Article • 02/13/2023

This article describes what is new in .NET Core 3.0. One of the biggest enhancements is support for Windows desktop applications (Windows only). By using the .NET Core 3.0 SDK component Windows Desktop, you can port your Windows Forms and Windows Presentation Foundation (WPF) applications. To be clear, the Windows Desktop component is only supported and included on Windows. For more information, see the [Windows desktop](#) section later in this article.

.NET Core 3.0 adds support for C# 8.0. It's highly recommended that you use [Visual Studio 2019 version 16.3](#) or newer, [Visual Studio for Mac 8.3](#) or newer, or [Visual Studio Code](#) with the latest [C# extension](#).

[Download and get started with .NET Core 3.0](#) right now on Windows, macOS, or Linux.

For more information about the release, see the [.NET Core 3.0 announcement](#).

.NET Core 3.0 RC 1 was considered production ready by Microsoft and was fully supported. If you're using a preview release, you must move to the RTM version for continued support.

Language improvements C# 8.0

C# 8.0 is also part of this release, which includes the [nullable reference types](#) feature, async streams, and more patterns. For more information about C# 8.0 features, see [What's new in C# 8.0](#).

Tutorials related to C# 8.0 language features:

- [Tutorial: Express your design intent more clearly with nullable and non-nullable reference types](#)
- [Tutorial: Generate and consume async streams using C# 8.0 and .NET Core 3.0](#)
- [Tutorial: Use pattern matching to build type-driven and data-driven algorithms](#)

Language enhancements were added to support the following API features detailed below:

- [Ranges and indices](#)
- [Async streams](#)

.NET Standard 2.1

.NET Core 3.0 implements .NET Standard 2.1. However, the default `dotnet new classlib` template generates a project that still targets .NET Standard 2.0. To target .NET Standard 2.1, edit your project file and change the `TargetFramework` property to `netstandard2.1`:

```
XML

<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <TargetFramework>netstandard2.1</TargetFramework>
  </PropertyGroup>

</Project>
```

If you're using Visual Studio, you need [Visual Studio 2019](#), as Visual Studio 2017 doesn't support .NET Standard 2.1 or .NET Core 3.0.

Compile/Deploy

Default executables

.NET Core now builds [framework-dependent executables](#) by default. This behavior is new for applications that use a globally installed version of .NET Core. Previously, only [self-contained deployments](#) would produce an executable.

During `dotnet build` or `dotnet publish`, an executable (known as the `appHost`) is created that matches the environment and platform of the SDK you're using. You can expect the same things with these executables as you would other native executables, such as:

- You can double-click on the executable.
- You can launch the application from a command prompt directly, such as `myapp.exe` on Windows, and `./myapp` on Linux and macOS.

macOS appHost and notarization

macOS only

Starting with the notarized .NET Core SDK 3.0 for macOS, the setting to produce a default executable (known as the appHost) is disabled by default. For more information, see [macOS Catalina Notarization and the impact on .NET Core downloads and projects](#).

When the appHost setting is enabled, .NET Core generates a native Mach-O executable when you build or publish. Your app runs in the context of the appHost when it is run from source code with the `dotnet run` command, or by starting the Mach-O executable directly.

Without the appHost, the only way a user can start a [framework-dependent](#) app is with the `dotnet <filename.dll>` command. An appHost is always created when you publish your app [self-contained](#).

You can either configure the appHost at the project level, or toggle the appHost for a specific `dotnet` command with the `-p:UseAppHost` parameter:

- Project file

XML

```
<PropertyGroup>
  <UseAppHost>true</UseAppHost>
</PropertyGroup>
```

- Command-line parameter

.NET CLI

```
dotnet run -p:UseAppHost=true
```

For more information about the `UseAppHost` setting, see [MSBuild properties for Microsoft.NET.Sdk](#).

Single-file executables

The `dotnet publish` command supports packaging your app into a platform-specific single-file executable. The executable is self-extracting and contains all dependencies (including native) that are required to run your app. When the app is first run, the application is extracted to a directory based on the app name and build identifier. Startup is faster when the application is run again. The application doesn't need to extract itself a second time unless a new version was used.

To publish a single-file executable, set the `PublishSingleFile` in your project or on the command line with the `dotnet publish` command:

XML

```
<PropertyGroup>
  <RuntimeIdentifier>win10-x64</RuntimeIdentifier>
  <PublishSingleFile>true</PublishSingleFile>
</PropertyGroup>
```

-or-

.NET CLI

```
dotnet publish -r win10-x64 -p:PublishSingleFile=true
```

For more information about single-file publishing, see the [single-file bundler design document](#).

Assembly trimming

The .NET core 3.0 SDK comes with a tool that can reduce the size of apps by analyzing IL and trimming unused assemblies.

Self-contained apps include everything needed to run your code, without requiring .NET to be installed on the host computer. However, many times the app only requires a small subset of the framework to function, and other unused libraries could be removed.

.NET Core now includes a setting that will use the [IL Trimmer](#) tool to scan the IL of your app. This tool detects what code is required, and then trims unused libraries. This tool can significantly reduce the deployment size of some apps.

To enable this tool, add the `<PublishTrimmed>` setting in your project and publish a self-contained app:

XML

```
<PropertyGroup>
  <PublishTrimmed>true</PublishTrimmed>
</PropertyGroup>
```

.NET CLI

```
dotnet publish -r <rid> -c Release
```

As an example, the basic "hello world" new console project template that is included, when published, hits about 70 MB in size. By using `<PublishTrimmed>`, that size is reduced to about 30 MB.

It's important to consider that applications or frameworks (including ASP.NET Core and WPF) that use reflection or related dynamic features, will often break when trimmed. This breakage occurs because the trimmer doesn't know about this dynamic behavior and can't determine which framework types are required for reflection. The IL Trimmer tool can be configured to be aware of this scenario.

Above all else, be sure to test your app after trimming.

For more information about the IL Trimmer tool, see the [documentation](#) or visit the [mono/linker](#) repo.

Tiered compilation

[Tiered compilation](#) (TC) is on by default with .NET Core 3.0. This feature enables the runtime to more adaptively use the just-in-time (JIT) compiler to achieve better performance.

The main benefit of tiered compilation is to provide two ways of jitting methods: in a lower-quality-but-faster tier or a higher-quality-but-slower tier. The quality refers to how well the method is optimized. TC helps to improve the performance of an application as it goes through various stages of execution, from startup through steady state. When tiered compilation is disabled, every method is compiled in a single way that's biased to steady-state performance over startup performance.

When TC is enabled, the following behavior applies for method compilation when an app starts up:

- If the method has ahead-of-time-compiled code, or [ReadyToRun](#), the pregenerated code is used.
- Otherwise, the method is jitted. Typically, these methods are generics over value types.
 - *Quick JIT* produces lower-quality (or less optimized) code more quickly. In .NET Core 3.0, Quick JIT is enabled by default for methods that don't contain loops and is preferred during startup.
 - The fully optimizing JIT produces higher-quality (or more optimized) code more slowly. For methods where Quick JIT would not be used (for example, if the

method is attributed with [MethodImplOptions.AggressiveOptimization](#)), the fully optimizing JIT is used.

For frequently called methods, the just-in-time compiler eventually creates fully optimized code in the background. The optimized code then replaces the pre-compiled code for that method.

Code generated by Quick JIT may run slower, allocate more memory, or use more stack space. If there are issues, you can disable Quick JIT using this MSBuild property in the project file:

XML

```
<PropertyGroup>
  <TieredCompilationQuickJit>false</TieredCompilationQuickJit>
</PropertyGroup>
```

To disable TC completely, use this MSBuild property in your project file:

XML

```
<PropertyGroup>
  <TieredCompilation>false</TieredCompilation>
</PropertyGroup>
```

💡 Tip

If you change these settings in the project file, you may need to perform a clean build for the new settings to be reflected (delete the `obj` and `bin` directories and rebuild).

For more information about configuring compilation at run time, see [Runtime configuration options for compilation](#).

ReadyToRun images

You can improve the startup time of your .NET Core application by compiling your application assemblies as ReadyToRun (R2R) format. R2R is a form of ahead-of-time (AOT) compilation.

R2R binaries improve startup performance by reducing the amount of work the just-in-time (JIT) compiler needs to do as your application loads. The binaries contain similar native code compared to what the JIT would produce. However, R2R binaries are larger

because they contain both intermediate language (IL) code, which is still needed for some scenarios, and the native version of the same code. R2R is only available when you publish a self-contained app that targets specific runtime environments (RID) such as Linux x64 or Windows x64.

To compile your project as ReadyToRun, do the following:

1. Add the `<PublishReadyToRun>` setting to your project:

XML

```
<PropertyGroup>
  <PublishReadyToRun>true</PublishReadyToRun>
</PropertyGroup>
```

2. Publish a self-contained app. For example, this command creates a self-contained app for the 64-bit version of Windows:

.NET CLI

```
dotnet publish -c Release -r win-x64 --self-contained
```

Cross platform/architecture restrictions

The ReadyToRun compiler doesn't currently support cross-targeting. You must compile on a given target. For example, if you want R2R images for Windows x64, you need to run the publish command on that environment.

Exceptions to cross-targeting:

- Windows x64 can be used to compile Windows Arm32, Arm64, and x86 images.
- Windows x86 can be used to compile Windows Arm32 images.
- Linux x64 can be used to compile Linux Arm32 and Arm64 images.

For more information, see [Ready to Run](#).

Runtime/SDK

Major-version runtime roll forward

.NET Core 3.0 introduces an opt-in feature that allows your app to roll forward to the latest major version of .NET Core. Additionally, a new setting has been added to control

how roll forward is applied to your app. This can be configured in the following ways:

- Project file property: `RollForward`
- Runtime configuration file property: `rollForward`
- Environment variable: `DOTNET_ROLL_FORWARD`
- Command-line argument: `--roll-forward`

One of the following values must be specified. If the setting is omitted, **Minor** is the default.

- **LatestPatch**
Roll forward to the highest patch version. This disables minor version roll forward.
- **Minor**
Roll forward to the lowest higher minor version, if requested minor version is missing. If the requested minor version is present, then the **LatestPatch** policy is used.
- **Major**
Roll forward to lowest higher major version, and lowest minor version, if requested major version is missing. If the requested major version is present, then the **Minor** policy is used.
- **LatestMinor**
Roll forward to highest minor version, even if requested minor version is present.
Intended for component hosting scenarios.
- **LatestMajor**
Roll forward to highest major and highest minor version, even if requested major is present. Intended for component hosting scenarios.
- **Disable**
Don't roll forward. Only bind to specified version. This policy isn't recommended for general use because it disables the ability to roll forward to the latest patches.
This value is only recommended for testing.

Besides the **Disable** setting, all settings will use the highest available patch version.

By default, if the requested version (as specified in `.runtimeconfig.json` for the application) is a release version, only release versions are considered for roll forward. Any pre-release versions are ignored. If there is no matching release version, then pre-release versions are taken into account. This behavior can be changed by setting `DOTNET_ROLL_FORWARD_TO_PRERELEASE=1`, in which case all versions are always considered.

Build copies dependencies

The `dotnet build` command now copies NuGet dependencies for your application from the NuGet cache to the build output folder. Previously, dependencies were only copied as part of `dotnet publish`.

There are some operations, like trimming and razor page publishing, that will still require publishing.

Local tools

.NET Core 3.0 introduces local tools. Local tools are similar to [global tools](#) but are associated with a particular location on disk. Local tools aren't available globally and are distributed as NuGet packages.

Local tools rely on a manifest file name `dotnet-tools.json` in your current directory. This manifest file defines the tools to be available at that folder and below. You can distribute the manifest file with your code to ensure that anyone who works with your code can restore and use the same tools.

For both global and local tools, a compatible version of the runtime is required. Many tools currently on NuGet.org target .NET Core Runtime 2.1. To install these tools globally or locally, you would still need to install the [NET Core 2.1 Runtime](#) ↗.

New `global.json` options

The `global.json` file has new options that provide more flexibility when you're trying to define which version of the .NET Core SDK is used. The new options are:

- `allowPrerelease`: Indicates whether the SDK resolver should consider prerelease versions when selecting the SDK version to use.
- `rollForward`: Indicates the roll-forward policy to use when selecting an SDK version, either as a fallback when a specific SDK version is missing or as a directive to use a higher version.

For more information about the changes including default values, supported values, and new matching rules, see [global.json overview](#).

Smaller Garbage Collection heap sizes

The Garbage Collector's default heap size has been reduced resulting in .NET Core using less memory. This change better aligns with the generation 0 allocation budget with modern processor cache sizes.

Garbage Collection Large Page support

Large Pages (also known as Huge Pages on Linux) is a feature where the operating system is able to establish memory regions larger than the native page size (often 4K) to improve performance of the application requesting these large pages.

The Garbage Collector can now be configured with the **GCLargePages** setting as an opt-in feature to choose to allocate large pages on Windows.

Windows Desktop & COM

.NET Core SDK Windows Installer

The MSI installer for Windows has changed starting with .NET Core 3.0. The SDK installers will now upgrade SDK feature-band releases in place. Feature bands are defined in the *hundreds* groups in the *patch* section of the version number. For example, **3.0.101** and **3.0.201** are versions in two different feature bands while **3.0.101** and **3.0.199** are in the same feature band. And, when .NET Core SDK **3.0.101** is installed, .NET Core SDK **3.0.100** will be removed from the machine if it exists. When .NET Core SDK **3.0.200** is installed on the same machine, .NET Core SDK **3.0.101** won't be removed.

For more information about versioning, see [Overview of how .NET Core is versioned](#).

Windows desktop

.NET Core 3.0 supports Windows desktop applications using Windows Presentation Foundation (WPF) and Windows Forms. These frameworks also support using modern controls and Fluent styling from the Windows UI XAML Library (WinUI) via [XAML islands](#).

The Windows Desktop component is part of the Windows .NET Core 3.0 SDK.

You can create a new WPF or Windows Forms app with the following `dotnet` commands:

.NET CLI

```
dotnet new wpf  
dotnet new winforms
```

Visual Studio 2019 adds **New Project** templates for .NET Core 3.0 Windows Forms and WPF.

For more information about how to port an existing .NET Framework application, see [Port WPF projects](#) and [Port Windows Forms projects](#).

WinForms high DPI

.NET Core Windows Forms applications can set high DPI mode with [Application.SetHighDpiMode\(HighDpiMode\)](#). The `SetHighDpiMode` method sets the corresponding high DPI mode unless the setting has been set by other means like `App.Manifest` or P/Invoke before `Application.Run`.

The possible `highDpiMode` values, as expressed by the [System.Windows.Forms.HighDpiMode](#) enum are:

- `DpiUnaware`
- `SystemAware`
- `PerMonitor`
- `PerMonitorV2`
- `DpiUnawareGdiScaled`

For more information about high DPI modes, see [High DPI Desktop Application Development on Windows](#).

Create COM components

On Windows, you can now create COM-callable managed components. This capability is critical to use .NET Core with COM add-in models and also to provide parity with .NET Framework.

Unlike .NET Framework where the *mscoree.dll* was used as the COM server, .NET Core will add a native launcher dll to the *bin* directory when you build your COM component.

For an example of how to create a COM component and consume it, see the [COM Demo](#).

Windows Native Interop

Windows offers a rich native API in the form of flat C APIs, COM, and WinRT. While .NET Core supports P/Invoke, .NET Core 3.0 adds the ability to [CoCreate COM APIs](#) and [Activate WinRT APIs](#). For a code example, see the [Excel Demo](#).

MSIX Deployment

[MSIX](#) is a new Windows application package format. It can be used to deploy .NET Core 3.0 desktop applications to Windows 10.

The [Windows Application Packaging Project](#), available in Visual Studio 2019, allows you to create MSIX packages with [self-contained](#) .NET Core applications.

The .NET Core project file must specify the supported runtimes in the `<RuntimeIdentifiers>` property:

XML

```
<RuntimeIdentifiers>win-x86;win-x64</RuntimeIdentifiers>
```

Linux improvements

SerialPort for Linux

.NET Core 3.0 provides basic support for [System.IO.Ports.SerialPort](#) on Linux.

Previously, .NET Core only supported using `SerialPort` on Windows.

For more information about the limited support for the serial port on Linux, see [GitHub issue #33146](#).

Docker and cgroup memory Limits

Running .NET Core 3.0 on Linux with Docker works better with cgroup memory limits. Running a Docker container with memory limits, such as with `docker run -m`, changes how .NET Core behaves.

- Default Garbage Collector (GC) heap size: maximum of 20 mb or 75% of the memory limit on the container.
- Explicit size can be set as an absolute number or percentage of cgroup limit.
- Minimum reserved segment size per GC heap is 16 mb. This size reduces the number of heaps that are created on machines.

GPIO Support for Raspberry Pi

Two packages have been released to NuGet that you can use for GPIO programming:

- [System.Device.Gpio](#)
- [IoT.Device.Bindings](#)

The GPIO packages include APIs for *GPIO*, *SPI*, *I2C*, and *PWM* devices. The IoT bindings package includes device bindings. For more information, see the [devices GitHub repo](#).

Arm64 Linux support

.NET Core 3.0 adds support for Arm64 for Linux. The primary use case for Arm64 is currently with IoT scenarios. For more information, see [.NET Core Arm64 Status](#).

Docker images for [.NET Core on Arm64](#) are available for Alpine, Debian, and Ubuntu.

ⓘ Note

Support for the macOS Arm64 (or "Apple Silicon") and Windows Arm64 operating systems was later added in .NET 6.

Security

TLS 1.3 & OpenSSL 1.1.1 on Linux

.NET Core now takes advantage of [TLS 1.3 support in OpenSSL 1.1.1](#), when it's available in a given environment. With TLS 1.3:

- Connection times are improved with reduced round trips required between the client and server.
- Improved security because of the removal of various obsolete and insecure cryptographic algorithms.

When available, .NET Core 3.0 uses [OpenSSL 1.1.1](#), [OpenSSL 1.1.0](#), or [OpenSSL 1.0.2](#) on a Linux system. When [OpenSSL 1.1.1](#) is available, both [System.Net.Security.SslStream](#) and [System.Net.Http.HttpClient](#) types will use [TLS 1.3](#) (assuming both the client and server support [TLS 1.3](#)).

ⓘ Important

Windows and macOS do not yet support [TLS 1.3](#).

The following C# 8.0 example demonstrates .NET Core 3.0 on Ubuntu 18.10 connecting to <https://www.cloudflare.com>:

```
C#
```

```

using System;
using System.Net.Security;
using System.Net.Sockets;
using System.Threading.Tasks;

namespace whats_new
{
    public static class TLS
    {
        public static async Task ConnectCloudFlare()
        {
            var targetHost = "www.cloudflare.com";

            using TcpClient tcpClient = new TcpClient();

            await tcpClient.ConnectAsync(targetHost, 443);

            using SslStream sslStream = new
SslStream(tcpClient.GetStream());

            await sslStream.AuthenticateAsClientAsync(targetHost);
            await Console.Out.WriteLineAsync($"Connected to {targetHost}
with {sslStream.SslProtocol}");
        }
    }
}

```

Cryptography ciphers

.NET Core 3.0 adds support for **AES-GCM** and **AES-CCM** ciphers, implemented with `System.Security.Cryptography.AesGcm` and `System.Security.Cryptography.AesCcm` respectively. These algorithms are both [Authenticated Encryption with Association Data \(AEAD\) algorithms](#).

The following code demonstrates using `AesGcm` cipher to encrypt and decrypt random data.

C#

```

using System;
using System.Linq;
using System.Security.Cryptography;

namespace whats_new
{
    public static class Cipher
    {
        public static void Run()
        {

```

```

        // key should be: pre-known, derived, or transported via another
channel, such as RSA encryption
        byte[] key = new byte[16];
        RandomNumberGenerator.Fill(key);

        byte[] nonce = new byte[12];
        RandomNumberGenerator.Fill(nonce);

        // normally this would be your data
        byte[] dataToEncrypt = new byte[1234];
        byte[] associatedData = new byte[333];
        RandomNumberGenerator.Fill(dataToEncrypt);
        RandomNumberGenerator.Fill(associatedData);

        // these will be filled during the encryption
        byte[] tag = new byte[16];
        byte[] ciphertext = new byte[dataToEncrypt.Length];

        using (AesGcm aesGcm = new AesGcm(key))
        {
            aesGcm.Encrypt(nonce, dataToEncrypt, ciphertext, tag,
associatedData);
        }

        // tag, nonce, ciphertext, associatedData should be sent to the
other part

        byte[] decryptedData = new byte[ciphertext.Length];

        using (AesGcm aesGcm = new AesGcm(key))
        {
            aesGcm.Decrypt(nonce, ciphertext, tag, decryptedData,
associatedData);
        }

        // do something with the data
        // this should always print that data is the same
        Console.WriteLine($"AES-GCM: Decrypted data is
{((dataToEncrypt.SequenceEqual(decryptedData) ? "the same as" : "different
than"))} original data.");
    }
}
}

```

Cryptographic Key Import/Export

.NET Core 3.0 supports the import and export of asymmetric public and private keys from standard formats. You don't need to use an X.509 certificate.

All key types, such as *RSA*, *DSA*, *ECDsa*, and *ECDiffieHellman*, support the following formats:

- **Public Key**
 - X.509 SubjectPublicKeyInfo
- **Private key**
 - PKCS#8 PrivateKeyInfo
 - PKCS#8 EncryptedPrivateKeyInfo

RSA keys also support:

- **Public Key**
 - PKCS#1 RSA PublicKey
- **Private key**
 - PKCS#1 RSA Private Key

The export methods produce DER-encoded binary data, and the import methods expect the same. If a key is stored in the text-friendly PEM format, the caller will need to base64-decode the content before calling an import method.

```
C#
using System;
using System.Security.Cryptography;

namespace whats_new
{
    public static class RSATest
    {
        public static void Run(string keyFile)
        {
            using var rsa = RSA.Create();

            byte[] keyBytes = System.IO.File.ReadAllBytes(keyFile);
            rsa.ImportRSAPrivateKey(keyBytes, out int bytesRead);

            Console.WriteLine($"Read {bytesRead} bytes, {keyBytes.Length - bytesRead} extra byte(s) in file.");
            RSAParameters rsaParameters = rsa.ExportParameters(true);
            Console.WriteLine(BitConverter.ToString(rsaParameters.D));
        }
    }
}
```

PKCS#8 files can be inspected with

[System.Security.Cryptography.Pkcs8PrivateKeyInfo](#) and PFX/PKCS#12 files can be inspected with [System.Security.Cryptography.Pkcs12Info](#). PFX/PKCS#12 files can be manipulated with [System.Security.Cryptography.Pkcs12Builder](#).

.NET Core 3.0 API changes

Ranges and indices

The new [System.Index](#) type can be used for indexing. You can create one from an `int` that counts from the beginning, or with a prefix `^` operator (C#) that counts from the end:

```
C#
```

```
Index i1 = 3; // number 3 from beginning
Index i2 = ^4; // number 4 from end
int[] a = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
Console.WriteLine($"{a[i1]}, {a[i2]}"); // "3, 6"
```

There's also the [System.Range](#) type, which consists of two `Index` values, one for the start and one for the end, and can be written with a `x..y` range expression (C#). You can then index with a `Range`, which produces a slice:

```
C#
```

```
var slice = a[i1..i2]; // { 3, 4, 5 }
```

For more information, see the [ranges and indices tutorial](#).

Async streams

The [IAsyncEnumerable<T>](#) type is a new asynchronous version of [IEnumerable<T>](#). The language lets you `await foreach` over [IAsyncEnumerable<T>](#) to consume their elements, and use `yield return` to them to produce elements.

The following example demonstrates both production and consumption of async streams. The `foreach` statement is async and itself uses `yield return` to produce an async stream for callers. This pattern (using `yield return`) is the recommended model for producing async streams.

```
C#
```

```
async IAsyncEnumerable<int> GetBigResultsAsync()
{
    await foreach (var result in GetResultsAsync())
    {
        if (result > 20) yield return result;
```

```
    }  
}
```

In addition to being able to `await foreach`, you can also create async iterators, for example, an iterator that returns an `IAsyncEnumerable/IAsyncEnumerator` that you can both `await` and `yield` in. For objects that need to be disposed, you can use `IAsyncDisposable`, which various BCL types implement, such as `Stream` and `Timer`.

For more information, see the [async streams tutorial](#).

IEEE Floating-point

Floating point APIs are being updated to comply with [IEEE 754-2008 revision](#). The goal of these changes is to expose all **required** operations and ensure that they're behaviorally compliant with the IEEE spec. For more information about floating-point improvements, see the [Floating-Point Parsing and Formatting improvements in .NET Core 3.0](#) blog post.

Parsing and formatting fixes include:

- Correctly parse and round inputs of any length.
- Correctly parse and format negative zero.
- Correctly parse `Infinity` and `Nan` by doing a case-insensitive check and allowing an optional preceding `+` where applicable.

New `System.Math` APIs include:

- `BitIncrement(Double)` and `BitDecrement(Double)`
Corresponds to the `nextUp` and `nextDown` IEEE operations. They return the smallest floating-point number that compares greater or lesser than the input (respectively). For example, `Math.BitIncrement(0.0)` would return `double.Epsilon`.
- `MaxMagnitude(Double, Double)` and `MinMagnitude(Double, Double)`
Corresponds to the `maxNumMag` and `minNumMag` IEEE operations, they return the value that is greater or lesser in magnitude of the two inputs (respectively). For example, `Math.MaxMagnitude(2.0, -3.0)` would return `-3.0`.
- `ILogB(Double)`
Corresponds to the `logB` IEEE operation that returns an integral value, it returns the integral base-2 log of the input parameter. This method is effectively the same as `floor(log2(x))`, but done with minimal rounding error.

- [ScaleB\(Double, Int32\)](#)

Corresponds to the `scaleB` IEEE operation that takes an integral value, it returns effectively `x * pow(2, n)`, but is done with minimal rounding error.

- [Log2\(Double\)](#)

Corresponds to the `log2` IEEE operation, it returns the base-2 logarithm. It minimizes rounding error.

- [FusedMultiplyAdd\(Double, Double, Double\)](#)

Corresponds to the `fma` IEEE operation, it performs a fused multiply add. That is, it does `(x * y) + z` as a single operation, thereby minimizing the rounding error. An example is `FusedMultiplyAdd(1e308, 2.0, -1e308)`, which returns `1e308`. The regular `(1e308 * 2.0) - 1e308` returns `double.PositiveInfinity`.

- [CopySign\(Double, Double\)](#)

Corresponds to the `copySign` IEEE operation, it returns the value of `x`, but with the sign of `y`.

.NET Platform-Dependent Intrinsics

APIs have been added that allow access to certain perf-oriented CPU instructions, such as the **SIMD** or **Bit Manipulation instruction** sets. These instructions can help achieve significant performance improvements in certain scenarios, such as processing data efficiently in parallel.

Where appropriate, the .NET libraries have begun using these instructions to improve performance.

For more information, see [.NET Platform-Dependent Intrinsics](#).

Improved .NET Core Version APIs

Starting with .NET Core 3.0, the version APIs provided with .NET Core now return the information you expect. For example:

```
C#
```

```
System.Console.WriteLine($"Environment.Version:  
{System.Environment.Version}");  
  
// Old result  
// Environment.Version: 4.0.30319.42000  
//
```

```
// New result  
// Environment.Version: 3.0.0
```

C#

```
System.Console.WriteLine($"RuntimeInformation.FrameworkDescription:  
{System.Runtime.InteropServices.RuntimeInformation.FrameworkDescription}");  
  
// Old result  
// RuntimeInformation.FrameworkDescription: .NET Core 4.6.27415.71  
//  
// New result (notice the value includes any preview release information)  
// RuntimeInformation.FrameworkDescription: .NET Core 3.0.0-preview4-  
27615-11
```

⚠ Warning

Breaking change. This is technically a breaking change because the versioning scheme has changed.

Fast built-in JSON support

.NET users have largely relied on [Newtonsoft.Json](#) and other popular JSON libraries, which continue to be good choices. `Newtonsoft.Json` uses .NET strings as its base datatype, which is UTF-16 under the hood.

The new built-in JSON support is high-performance, low allocation, and works with UTF-8 encoded JSON text. For more information about the `System.Text.Json` namespace and types, see the following articles:

- [JSON serialization in .NET - overview](#)
- [How to serialize and deserialize JSON in .NET.](#)
- [How to migrate from Newtonsoft.Json to System.Text.Json](#)

HTTP/2 support

The `System.Net.Http.HttpClient` type supports the HTTP/2 protocol. If HTTP/2 is enabled, the HTTP protocol version is negotiated via TLS/ALPN, and HTTP/2 is used if the server elects to use it.

The default protocol remains HTTP/1.1, but HTTP/2 can be enabled in two different ways. First, you can set the HTTP request message to use HTTP/2:

C#

```
var client = new HttpClient() { BaseAddress = new Uri("https://localhost:5001") };

// HTTP/1.1 request
using (var response = await client.GetAsync("/"))
    Console.WriteLine(response.Content);

// HTTP/2 request
using (var request = new HttpRequestMessage(HttpMethod.Get, "/") { Version =
    new Version(2, 0) })
    using (var response = await client.SendAsync(request))
        Console.WriteLine(response.Content);
```

Second, you can change [HttpClient](#) to use HTTP/2 by default:

C#

```
var client = new HttpClient()
{
    BaseAddress = new Uri("https://localhost:5001"),
    DefaultRequestVersion = new Version(2, 0)
};

// HTTP/2 is default
using (var response = await client.GetAsync("/"))
    Console.WriteLine(response.Content);
```

Many times when you're developing an application, you want to use an unencrypted connection. If you know the target endpoint will be using HTTP/2, you can turn on unencrypted connections for HTTP/2. You can turn it on by setting the `DOTNET_SYSTEM_NET_HTTP_SOCKETSHTTPHANDLER_HTTP2UNENCRYPTEDSUPPORT` environment variable to `1` or by enabling it in the app context:

C#

```
ApplicationContext.SetSwitch("System.Net.Http.SocketsHttpHandler.Http2UnencryptedSupport", true);
```

Next steps

- [Review the breaking changes between .NET Core 2.2 and 3.0.](#)
- [Review the breaking changes in .NET Core 3.0 for Windows Forms apps.](#)

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

Breaking changes in .NET Core 3.0

Article • 03/18/2023

If you're migrating to version 3.0 of .NET Core, ASP.NET Core, or EF Core, the breaking changes listed in this article may affect your app.

ASP.NET Core

- Obsolete Antiforgery, CORS, Diagnostics, MVC, and Routing APIs removed
- "Pubternal" APIs removed
- Authentication: Google+ deprecation
- Authentication: `HttpContext.Authentication` property removed
- Authentication: `Newtonsoft.Json` types replaced
- Authentication: `OAuthHandler` `ExchangeCodeAsync` signature changed
- Authorization: `AddAuthorization` overload moved to different assembly
- Authorization: `IAllowAnonymous` removed from `AuthorizationFilterContext.Filters`
- Authorization: `IAuthorizationPolicyProvider` implementations require new method
- Caching: `CompactOnMemoryPressure` property removed
- Caching: `Microsoft.Extensions.Caching.SqlServer` uses new `SqlClient` package
- Caching: `ResponseCaching` "pubternal" types changed to internal
- Data Protection: `DataProtection.Blobs` uses new Azure Storage APIs
- Hosting: `AspNetCoreModule V1` removed from Windows Hosting Bundle
- Hosting: Generic host restricts Startup constructor injection
- Hosting: HTTPS redirection enabled for IIS out-of-process apps
- Hosting: `IHostingEnvironment` and `IAplicationLifetime` types replaced
- Hosting: `ObjectPoolProvider` removed from `WebHostBuilder` dependencies
- HTTP: `DefaultHttpContext` extensibility removed
- HTTP: `HeaderNames` fields changed to static readonly
- HTTP: Response body infrastructure changes
- HTTP: Some cookie SameSite default values changed
- HTTP: Synchronous IO disabled by default
- Identity: `AddDefaultUI` method overload removed
- Identity: UI Bootstrap version change
- Identity: `SignInAsync` throws exception for unauthenticated identity
- Identity: `SignInManager` constructor accepts new parameter
- Identity: UI uses static web assets feature
- Kestrel: Connection adapters removed
- Kestrel: Empty HTTPS assembly removed
- Kestrel: Request trailer headers moved to new collection

- Kestrel: Transport abstraction layer changes
- Localization: APIs marked obsolete
- Logging: DebugLogger class made internal
- MVC: Controller action Async suffix removed
- MVC: JsonResult moved to Microsoft.AspNetCore.Mvc.Core
- MVC: Precompilation tool deprecated
- MVC: Types changed to internal
- MVC: Web API compatibility shim removed
- Razor: RazorTemplateEngine API removed
- Razor: Runtime compilation moved to a package
- Session state: Obsolete APIs removed
- Shared framework: Assembly removal from Microsoft.AspNetCore.App
- Shared framework: Microsoft.AspNetCore.All removed
- SignalR: HandshakeProtocol.SuccessHandshakeData replaced
- SignalR: HubConnection methods removed
- SignalR: HubConnectionContext constructors changed
- SignalR: JavaScript client package name change
- SignalR: Obsolete APIs
- SPAs: SpaServices and NodeServices marked obsolete
- SPAs: SpaServices and NodeServices console logger fallback default change
- Target framework: .NET Framework not supported

Obsolete Antiforgery, CORS, Diagnostics, MVC, and Routing APIs removed

Obsolete members and compatibility switches in ASP.NET Core 2.2 were removed.

Version introduced

3.0

Reason for change

Improvement of API surface over time.

Recommended action

While targeting .NET Core 2.2, follow the guidance in the obsolete build messages to adopt new APIs instead.

Category

ASP.NET Core

Affected APIs

The following types and members were marked as obsolete for ASP.NET Core 2.1 and 2.2:

Types

- `Microsoft.AspNetCore.Diagnostics.Views.WelcomePage`
- `Microsoft.AspNetCore.DiagnosticsViewPage.Views.AttributeValue`
- `Microsoft.AspNetCore.DiagnosticsViewPage.Views.BaseView`
- `Microsoft.AspNetCore.DiagnosticsViewPage.Views.HelperResult`
- `Microsoft.AspNetCore.Mvc.Formatters.Xml.ProblemDetails21Wrapper`
- `Microsoft.AspNetCore.Mvc.Formatters.Xml.ValidationProblemDetails21Wrapper`
- `Microsoft.AspNetCore.Mvc.Razor.Compilation.ViewsFeatureProvider`
- `Microsoft.AspNetCore.Mvc.RazorPages.Infrastructure.PageArgumentBinder`
- `Microsoft.AspNetCore.Routing.IRouteValuesAddressMetadata`
- `Microsoft.AspNetCore.Routing.RouteValuesAddressMetadata`

Constructors

- `Microsoft.AspNetCore.Cors.Infrastructure.CorsService(IOptions{CorsOptions})`
- `Microsoft.AspNetCore.Routing.Tree.TreeRouteBuilder(ILoggerFactory, UrlEncoder, ObjectPool{UriBuildingContext}, IInlineConstraintResolver)`
- `Microsoft.AspNetCore.Mvc.Formatters.OutputFormatterCanWriteContext`
- `Microsoft.AspNetCore.Mvc.ApiExplorer.DefaultApiDescriptionProvider(IOptions{MvcOptions}, IInlineConstraintResolver, IModelDataProvider)`
- `Microsoft.AspNetCore.Mvc.ApiExplorer.DefaultApiDescriptionProvider(IOptions{MvcOptions}, IInlineConstraintResolver, IModelDataProvider, IActionResultTypeMapper)`
- `Microsoft.AspNetCore.Mvc.Formatters.FormatFilter(IOptions{MvcOptions})`
- `Microsoft.AspNetCore.Mvc.ModelBinding.Binders.ArrayModelBinder`1(IModelBinder)`
- `Microsoft.AspNetCore.Mvc.ModelBinding.Binders.ByteArrayModelBinder`
- `Microsoft.AspNetCore.Mvc.ModelBinding.Binders.CollectionModelBinder`1(IModelBinder)`

- [Microsoft.AspNetCore.Mvc.ModelBinding.Binders.ComplexTypeModelBinder\(IDictionary`2\)](#)
- [Microsoft.AspNetCore.Mvc.ModelBinding.Binders.DictionaryModelBinder`2\(IModelBinder, IModelBinder\)](#)
- [Microsoft.AspNetCore.Mvc.ModelBinding.Binders.DoubleModelBinder\(System.Globalization.NumberStyles\)](#)
- [Microsoft.AspNetCore.Mvc.ModelBinding.Binders.FloatModelBinder\(System.Globalization.NumberStyles\)](#)
- [Microsoft.AspNetCore.Mvc.ModelBinding.Binders.FormCollectionModelBinder](#)
- [Microsoft.AspNetCore.Mvc.ModelBinding.Binders.FormFileModelBinder](#)
- [Microsoft.AspNetCore.Mvc.ModelBinding.Binders.HeaderModelBinder](#)
- [Microsoft.AspNetCore.Mvc.ModelBinding.Binders.KeyValuePairModelBinder`2\(IModelBinder, IModelBinder\)](#)
- [Microsoft.AspNetCore.Mvc.ModelBinding.Binders.SimpleTypeModelBinder\(System.Type\)](#)
- [Microsoft.AspNetCore.Mvc.ModelBinding.ModelAttributes\(IEnumerable{System.Object}\)](#)
- [Microsoft.AspNetCore.Mvc.ModelBinding.ModelAttributes\(IEnumerable{System.Object}, IEnumerable{System.Object}\)](#)
- [Microsoft.AspNetCore.Mvc.ModelBinding.ModelBinderFactory\(IModelMetadataProvider, IOptions{MvcOptions}\)](#)
- [Microsoft.AspNetCore.Mvc.ModelBinding.ParameterBinder\(IModelMetadataProvider, IModelBinderFactory, IObjectModelValidator\)](#)
- [**Microsoft.AspNetCore.Mvc.Routing.KnownRouteValueConstraint\(\)**](#)
- [Microsoft.AspNetCore.Mvc.Formatters.XmlDataContractSerializerInputFormatter](#)
- [Microsoft.AspNetCore.Mvc.Formatters.XmlDataContractSerializerInputFormatter\(System.Boolean\)](#)
- [Microsoft.AspNetCore.Mvc.Formatters.XmlDataContractSerializerInputFormatter\(MvcOptions\)](#)
- [Microsoft.AspNetCore.Mvc.Formatters.XmlSerializerInputFormatter](#)
- [Microsoft.AspNetCore.Mvc.Formatters.XmlSerializerInputFormatter\(System.Boolean\)](#)
- [Microsoft.AspNetCore.Mvc.Formatters.XmlSerializerInputFormatter\(MvcOptions\)](#)
- [**Microsoft.AspNetCore.Mvc.TagHelpers.ImageTagHelper\(IHostingEnvironment, IMemoryCache, HtmlEncoder, IUrlHelperFactory\)**](#)
- [Microsoft.AspNetCore.Mvc.TagHelpers.LinkTagHelper\(IHostingEnvironment, IMemoryCache, HtmlEncoder, JavaScriptEncoder, IUrlHelperFactory\)](#)

- Microsoft.AspNetCore.Mvc.TagHelpers.ScriptTagHelper(IHostingEnvironment, IMemoryCache, HtmlEncoder, JavaScriptEncoder, IUrlHelperFactory)
- Microsoft.AspNetCore.Mvc.RazorPages.Infrastructure.RazorPageAdapter(RazorPageBase)

Properties

- Microsoft.AspNetCore.Antiforgery.AntiforgeryOptions.CookieDomain
- Microsoft.AspNetCore.Antiforgery.AntiforgeryOptions.CookieName
- Microsoft.AspNetCore.Antiforgery.AntiforgeryOptions.CookiePath
- Microsoft.AspNetCore.Antiforgery.AntiforgeryOptions.RequireSsl
- Microsoft.AspNetCore.Mvc.ApiBehaviorOptions.AllowInferringBindingSourceForCollectionTypesAsFromQuery
- Microsoft.AspNetCore.Mvc.ApiBehaviorOptions.SuppressUseValidationProblemDetailsForInvalidModelStateResponses
- Microsoft.AspNetCore.Mvc.CookieTempDataProviderOptions.CookieName
- Microsoft.AspNetCore.Mvc.CookieTempDataProviderOptions.Domain
- Microsoft.AspNetCore.Mvc.CookieTempDataProviderOptions.Path
- Microsoft.AspNetCore.Mvc.DataAnnotations.MvcDataAnnotationsLocalizationOptions.AllowDataAnnotationsLocalizationForEnumDisplayAttributes
- Microsoft.AspNetCore.Mvc.Formatters.Xml.MvcXmlOptions.AllowRfc7807CompliantFormatDetails
- Microsoft.AspNetCore.Mvc.MvcOptions.AllowBindingHeaderValueToNonStringModelType
- Microsoft.AspNetCore.Mvc.MvcOptions.AllowCombiningAuthorizeFilters
- Microsoft.AspNetCore.Mvc.MvcOptions.AllowShortCircuitingValidationWhenNoValidatorsArePresent
- Microsoft.AspNetCore.Mvc.MvcOptions.AllowValidatingTopLevelNodes
- Microsoft.AspNetCore.Mvc.MvcOptions.InputFormatterExceptionPolicy
- Microsoft.AspNetCore.Mvc.MvcOptions.SuppressBindingUndefinedValueToEnumType
- Microsoft.AspNetCore.Mvc.MvcViewOptions.AllowRenderingMaxLengthAttribute
- Microsoft.AspNetCore.Mvc.MvcViewOptions.Suppress TempDataAttributePrefix
- Microsoft.AspNetCore.Mvc.RazorPages.RazorPagesOptions.AllowAreas
- Microsoft.AspNetCore.Mvc.RazorPages.RazorPagesOptions.AllowDefaultHandlingForOptionsRequests
- Microsoft.AspNetCore.Mvc.RazorPages.RazorPagesOptions.AllowMappingHeadRequestsToGetHandler

Methods

- `Microsoft.AspNetCore.Mvc.LocalRedirectResult.ExecuteResult(ActionContext)`
 - `Microsoft.AspNetCore.Mvc.RedirectResult.ExecuteResult(ActionContext)`
 - `Microsoft.AspNetCore.Mvc.RedirectToActionResult.ExecuteResult(ActionContext)`
 - `Microsoft.AspNetCore.Mvc.RedirectToPageResult.ExecuteResult(ActionContext)`
 - `Microsoft.AspNetCore.Mvc.RedirectToRouteResult.ExecuteResult(ActionContext)`
 - `Microsoft.AspNetCore.Mvc.ModelBinding.ParameterBinder.BindModelAsync(ActionContext, IValueProvider, ParameterDescriptor)`
 - `Microsoft.AspNetCore.Mvc.ModelBinding.ParameterBinder.BindModelAsync(ActionContext, IValueProvider, ParameterDescriptor, Object)`
-

"Pubternal" APIs removed

To better maintain the public API surface of ASP.NET Core, most of the types in `*.Internal` namespaces (referred to as "pubternal" APIs) have become truly internal. Members in these namespaces were never meant to be supported as public-facing APIs. The APIs could break in minor releases and often did. Code that depends on these APIs breaks when updating to ASP.NET Core 3.0.

For more information, see [dotnet/aspnetcore#4932](#) and [dotnet/aspnetcore#11312](#).

Version introduced

3.0

Old behavior

The affected APIs are marked with the `public` access modifier and exist in `*.Internal` namespaces.

New behavior

The affected APIs are marked with the `internal` access modifier and can no longer be used by code outside that assembly.

Reason for change

The guidance for these "pubternal" APIs was that they:

- Could change without notice.
- Weren't subject to .NET policies to prevent breaking changes.

Leaving the APIs `public` (even in the `*.Internal` namespaces) was confusing to customers.

Recommended action

Stop using these "pubternal" APIs. If you have questions about alternate APIs, open an issue in the [dotnet/aspnetcore](#) repository.

For example, consider the following HTTP request buffering code in an ASP.NET Core 2.2 project. The `EnableRewind` extension method exists in the `Microsoft.AspNetCore.Http.Internal` namespace.

C#

```
HttpContext.Request.EnableRewind();
```

In an ASP.NET Core 3.0 project, replace the `EnableRewind` call with a call to the `EnableBuffering` extension method. The request buffering feature works as it did in the past. `EnableBuffering` calls the now `internal` API.

C#

```
HttpContext.Request.EnableBuffering();
```

Category

ASP.NET Core

Affected APIs

All APIs in the `Microsoft.AspNetCore.*` and `Microsoft.Extensions.*` namespaces that have an `Internal` segment in the namespace name. For example:

- `Microsoft.AspNetCore.Authentication.Internal`
- `Microsoft.AspNetCore.Builder.Internal`
- `Microsoft.AspNetCore.DataProtection.Cng.Internal`
- `Microsoft.AspNetCore.DataProtection.Internal`
- `Microsoft.AspNetCore.Hosting.Internal`
- `Microsoft.AspNetCore.Http.Internal`
- `Microsoft.AspNetCore.Mvc.Core.Infrastructure`

- Microsoft.AspNetCore.Mvc.Core.Internal
 - Microsoft.AspNetCore.Mvc.Cors.Internal
 - Microsoft.AspNetCore.Mvc.DataAnnotations.Internal
 - Microsoft.AspNetCore.Mvc.Formatters.Internal
 - Microsoft.AspNetCore.Mvc.Formatters.Json.Internal
 - Microsoft.AspNetCore.Mvc.Formatters.Xml.Internal
 - Microsoft.AspNetCore.Mvc.Internal
 - Microsoft.AspNetCore.Mvc.ModelBinding.Internal
 - Microsoft.AspNetCore.Mvc.Razor.Internal
 - Microsoft.AspNetCore.Mvc.RazorPages.Internal
 - Microsoft.AspNetCore.Mvc.TagHelpers.Internal
 - Microsoft.AspNetCore.Mvc.ViewFeatures.Internal
 - Microsoft.AspNetCore.Rewrite.Internal
 - Microsoft.AspNetCore.Routing.Internal
 - Microsoft.AspNetCore.Server.Kestrel.Core.Adapter.Internal
 - Microsoft.AspNetCore.Server.Kestrel.Core.Internal.Http
 - Microsoft.AspNetCore.Server.Kestrel.Core.Internal.Infrastructure
 - Microsoft.AspNetCore.Server.Kestrel.Https.Internal
-

Authentication: Google+ deprecated and replaced

Google is starting to [shut down](#) Google+ Sign-in for apps as early as January 28, 2019.

Change description

ASP.NET 4.x and ASP.NET Core have been using the Google+ Sign-in APIs to authenticate Google account users in web apps. The affected NuGet packages are [Microsoft.AspNetCore.Authentication.Google](#) for ASP.NET Core and [Microsoft.Owin.Security.Google](#) for [Microsoft.Owin](#) with ASP.NET Web Forms and MVC.

Google's replacement APIs use a different data source and format. The mitigations and solutions provided below account for the structural changes. Apps should verify the data itself still satisfies their requirements. For example, names, email addresses, profile links, and profile photos may provide subtly different values than before.

Version introduced

All versions. This change is external to ASP.NET Core.

Recommended action

Owin with ASP.NET Web Forms and MVC

For `Microsoft.Owin` 3.1.0 and later, a temporary mitigation is outlined [here](#). Apps should complete testing with the mitigation to check for changes in the data format. There are plans to release `Microsoft.Owin` 4.0.1 with a fix. Apps using any prior version should update to version 4.0.1.

ASP.NET Core 1.x

The mitigation in [Owin with ASP.NET Web Forms and MVC](#) can be adapted to ASP.NET Core 1.x. NuGet package patches aren't planned because 1.x has reached [end of life](#) status.

ASP.NET Core 2.x

For `Microsoft.AspNetCore.Authentication.Google` version 2.x, replace your existing call to `AddGoogle` in `Startup.ConfigureServices` with the following code:

C#

```
.AddGoogle(o =>
{
    o.ClientId = Configuration["Authentication:Google:ClientId"];
    o.ClientSecret = Configuration["Authentication:Google:ClientSecret"];
    o.UserInformationEndpoint =
"https://www.googleapis.com/oauth2/v2/userinfo";
    o.ClaimActions.Clear();
    o.ClaimActions.MapJsonKey(ClaimTypes.NameIdentifier, "id");
    o.ClaimActions.MapJsonKey(ClaimTypes.Name, "name");
    o.ClaimActions.MapJsonKey(ClaimTypes.GivenName, "given_name");
    o.ClaimActions.MapJsonKey(ClaimTypes.Surname, "family_name");
    o.ClaimActions.MapJsonKey("urn:google:profile", "link");
    o.ClaimActions.MapJsonKey(ClaimTypes.Email, "email");
});
```

The February 2.1 and 2.2 patches incorporated the preceding reconfiguration as the new default. No patch is planned for ASP.NET Core 2.0 since it has reached [end of life](#).

ASP.NET Core 3.0

The mitigation given for ASP.NET Core 2.x can also be used for ASP.NET Core 3.0. In future 3.0 previews, the `Microsoft.AspNetCore.Authentication.Google` package may be removed. Users would be directed to `Microsoft.AspNetCore.Authentication.OpenIdConnect` instead. The following code shows how to replace `AddGoogle` with `AddOpenIdConnect` in `Startup.ConfigureServices`. This replacement can be used with ASP.NET Core 2.0 and later and can be adapted for ASP.NET Core 1.x as needed.

C#

```
.AddOpenIdConnect("Google", o =>
{
    o.ClientId = Configuration["Authentication:Google:ClientId"];
    o.ClientSecret = Configuration["Authentication:Google:ClientSecret"];
    o.Authority = "https://accounts.google.com";
    o.ResponseType = OpenIdConnectResponseType.Code;
    o.CallbackPath = "/signin-google"; // Or register the default "/signin-oidc"
    o.Scope.Add("email");
});
JwtSecurityTokenHandler.DefaultInboundClaimTypeMap.Clear();
```

Category

ASP.NET Core

Affected APIs

[Microsoft.AspNetCore.Authentication.Google](#)

Authentication: HttpContext.Authentication property removed

The deprecated `Authentication` property on `HttpContext` has been removed.

Change description

As part of [dotnet/aspnetcore#6504](#), the deprecated `Authentication` property on `HttpContext` has been removed. The `Authentication` property has been deprecated since 2.0. A [migration guide](#) was published to migrate code using this deprecated property to the new replacement APIs. The remaining unused classes / APIs related to

the old ASP.NET Core 1.x authentication stack were removed in commit [dotnet/aspnetcore@d7a7c65](#).

For discussion, see [dotnet/aspnetcore#6533](#).

Version introduced

3.0

Reason for change

ASP.NET Core 1.0 APIs have been replaced by extension methods in [Microsoft.AspNetCore.Authentication.AuthenticationHttpContextExtensions](#).

Recommended action

See the [migration guide](#).

Category

ASP.NET Core

Affected APIs

- [Microsoft.AspNetCore.Http.Authentication.AuthenticateInfo](#)
- [Microsoft.AspNetCore.Http.Authentication.AuthenticationManager](#)
- [Microsoft.AspNetCore.Http.Authentication.AuthenticationProperties](#)
- [Microsoft.AspNetCore.Http.Features.Authentication.AuthenticateContext](#)
- [Microsoft.AspNetCore.Http.Features.Authentication.ChallengeBehavior](#)
- [Microsoft.AspNetCore.Http.Features.Authentication.ChallengeContext](#)
- [Microsoft.AspNetCore.Http.Features.Authentication.DescribeSchemesContext](#)
- [Microsoft.AspNetCore.Http.Features.Authentication.IAuthenticationHandler](#)
- [Microsoft.AspNetCore.Http.Features.Authentication.IHttpAuthenticationFeature.Handler](#)
- [Microsoft.AspNetCore.Http.Features.Authentication.SignInContext](#)
- [Microsoft.AspNetCore.Http.Features.Authentication.SignOutContext](#)
- [Microsoft.AspNetCore.Http.HttpContext.Authentication](#)

Authentication: Newtonsoft.Json types replaced

In ASP.NET Core 3.0, `Newtonsoft.Json` types used in Authentication APIs have been replaced with `System.Text.Json` types. Except for the following cases, basic usage of the Authentication packages remains unaffected:

- Classes derived from the OAuth providers, such as those from [aspnet-contrib](#).
- Advanced claim manipulation implementations.

For more information, see [dotnet/aspnetcore#7105](#). For discussion, see [dotnet/aspnetcore#7289](#).

Version introduced

3.0

Recommended action

For derived OAuth implementations, the most common change is to replace `JObject.Parse` with `JsonDocument.Parse` in the `CreateTicketAsync` override as shown [here](#). `JsonDocument` implements `IDisposable`.

The following list outlines known changes:

- `ClaimAction.Run(JObject, ClaimsIdentity, String)` becomes `ClaimAction.Run(JsonElement userData, ClaimsIdentity identity, string issuer)`. All derived implementations of `ClaimAction` are similarly affected.
- `ClaimActionCollectionMapExtensions.MapCustomJson(ClaimActionCollection, String, Func< JObject, String >)` becomes `MapCustomJson(this ClaimActionCollection collection, string claimType, Func< JsonElement, string > resolver)`
- `ClaimActionCollectionMapExtensions.MapCustomJson(ClaimActionCollection, String, String, Func< JObject, String >)` becomes `MapCustomJson(this ClaimActionCollection collection, string claimType, string valueType, Func< JsonElement, string > resolver)`
- `OAuthCreatingTicketContext` has had one old constructor removed and the other replaced `JObject` with `JsonElement`. The `User` property and `RunClaimActions` method have been updated to match.
- `Success(JObject)` now accepts a parameter of type `JsonDocument` instead of `JObject`. The `Response` property has been updated to match. `OAuthTokenResponse` is now disposable and will be disposed by `OAuthHandler`. Derived OAuth implementations overriding `ExchangeCodeAsync` don't need to dispose the `JsonDocument` or `OAuthTokenResponse`.

- `UserInformationReceivedContext.User` changed from `JObject` to `JsonDocument`.
- `TwitterCreatingTicketContext.User` changed from `JObject` to `JsonElement`.
- The last parameter of `TwitterHandler.CreateTicketAsync(ClaimsIdentity, AuthenticationProperties, AccessToken, JObject)` changed from `JObject` to `JsonElement`. The replacement method is `TwitterHandler.CreateTicketAsync(ClaimsIdentity, AuthenticationProperties, AccessToken, JsonElement)`.

Category

ASP.NET Core

Affected APIs

- `Microsoft.AspNetCore.Authentication.Facebook`
- `Microsoft.AspNetCore.Authentication.Google`
- `Microsoft.AspNetCore.Authentication.MicrosoftAccount`
- `Microsoft.AspNetCore.Authentication.OAuth`
- `Microsoft.AspNetCore.Authentication.OpenIdConnect`
- `Microsoft.AspNetCore.Authentication.Twitter`

Authentication: OAuthHandler ExchangeCodeAsync signature changed

In ASP.NET Core 3.0, the signature of `OAuthHandler.ExchangeCodeAsync` was changed from:

C#

```
protected virtual
System.Threading.Tasks.Task<Microsoft.AspNetCore.Authentication.OAuth.OAuthT
okenResponse> ExchangeCodeAsync(string code, string redirectUri) { throw
null; }
```

To:

C#

```
protected virtual
System.Threading.Tasks.Task<Microsoft.AspNetCore.Authentication.OAuth.OAuthT
okenResponse>
```

```
ExchangeCodeAsync(Microsoft.AspNetCore.Authentication.OAuth.OAuthCodeExchangeContext context) { throw null; }
```

Version introduced

3.0

Old behavior

The `code` and `redirectUri` strings were passed as separate arguments.

New behavior

`Code` and `RedirectUri` are properties on `OAuthCodeExchangeContext` that can be set via the `OAuthCodeExchangeContext` constructor. The new `OAuthCodeExchangeContext` type is the only argument passed to `OAuthHandler.ExchangeCodeAsync`.

Reason for change

This change allows additional parameters to be provided in a non-breaking manner. There's no need to create new `ExchangeCodeAsync` overloads.

Recommended action

Construct an `OAuthCodeExchangeContext` with the appropriate `code` and `redirectUri` values. An `AuthenticationProperties` instance must be provided. This single `OAuthCodeExchangeContext` instance can be passed to `OAuthHandler.ExchangeCodeAsync` instead of multiple arguments.

Category

ASP.NET Core

Affected APIs

[OAuthHandler<TOptions>.ExchangeCodeAsync\(String, String\)](#)

Authorization: AddAuthorization overload moved to different assembly

The core `AddAuthorization` methods that used to reside in `Microsoft.AspNetCore.Authorization` were renamed to `AddAuthorizationCore`. The old `AddAuthorization` methods still exist, but are in the `Microsoft.AspNetCore.Authorization.Policy` assembly instead. Apps using both methods should see no impact. Note that `Microsoft.AspNetCore.Authorization.Policy` now ships in the shared framework rather than a standalone package as discussed in [Shared framework: Assemblies removed from Microsoft.AspNetCore.App](#).

Version introduced

3.0

Old behavior

`AddAuthorization` methods existed in `Microsoft.AspNetCore.Authorization`.

New behavior

`AddAuthorization` methods exist in `Microsoft.AspNetCore.Authorization.Policy`.

`AddAuthorizationCore` is the new name for the old methods.

Reason for change

`AddAuthorization` is a better method name for adding all common services needed for authorization.

Recommended action

Either add a reference to `Microsoft.AspNetCore.Authorization.Policy` or use `AddAuthorizationCore` instead.

Category

ASP.NET Core

Affected APIs

[Microsoft.Extensions.DependencyInjection.AuthorizationServiceCollectionExtensions.AddAuthorization\(IServiceCollection, Action<AuthorizationOptions>\)](#)

Authorization: `IAllowAnonymous` removed from `AuthorizationFilterContext.Filters`

As of ASP.NET Core 3.0, MVC doesn't add `AllowAnonymousFilters` for `[AllowAnonymous]` attributes that were discovered on controllers and action methods. This change is addressed locally for derivatives of `AuthorizeAttribute`, but it's a breaking change for `IAsyncAuthorizationFilter` and `IAuthorizationFilter` implementations. Such implementations wrapped in a `[TypeFilter]` attribute are a [popular ↗](#) and supported way to achieve strongly-typed, attribute-based authorization when both configuration and dependency injection are required.

Version introduced

3.0

Old behavior

`IAllowAnonymous` appeared in the `AuthorizationFilterContext.Filters` collection. Testing for the interface's presence was a valid approach to override or disable the filter on individual controller methods.

New behavior

`IAllowAnonymous` no longer appears in the `AuthorizationFilterContext.Filters` collection. `IAsyncAuthorizationFilter` implementations that are dependent on the old behavior typically cause intermittent HTTP 401 Unauthorized or HTTP 403 Forbidden responses.

Reason for change

A new endpoint routing strategy was introduced in ASP.NET Core 3.0.

Recommended action

Search the endpoint metadata for `IAllowAnonymous`. For example:

C#

```
var endpoint = context.HttpContext.GetEndpoint();
if (endpoint?.Metadata?.GetMetadata<IAuthorizationFilter>() != null)
```

```
{  
}
```

An example of this technique is seen in [this HasAllowAnonymous method ↗](#).

Category

ASP.NET Core

Affected APIs

None

Authorization: `IAuthorizationPolicyProvider` implementations require new method

In ASP.NET Core 3.0, a new `GetFallbackPolicyAsync` method was added to `IAuthorizationPolicyProvider`. This fallback policy is used by the authorization middleware when no policy is specified.

For more information, see [dotnet/aspnetcore#9759 ↗](#).

Version introduced

3.0

Old behavior

Implementations of `IAuthorizationPolicyProvider` didn't require a `GetFallbackPolicyAsync` method.

New behavior

Implementations of `IAuthorizationPolicyProvider` require a `GetFallbackPolicyAsync` method.

Reason for change

A new method was needed for the new `AuthorizationMiddleware` to use when no policy is specified.

Recommended action

Add the `GetFallbackPolicyAsync` method to your implementations of `IAuthorizationPolicyProvider`.

Category

ASP.NET Core

Affected APIs

[Microsoft.AspNetCore.Authorization.IAuthorizationPolicyProvider](#)

Caching: `CompactOnMemoryPressure` property removed

The ASP.NET Core 3.0 release removed the [obsolete MemoryCacheOptions APIs](#).

Change description

This change is a follow-up to [aspnet/Caching#221](#). For discussion, see [dotnet/extensions#1062](#).

Version introduced

3.0

Old behavior

`MemoryCacheOptions.CompactOnMemoryPressure` property was available.

New behavior

The `MemoryCacheOptions.CompactOnMemoryPressure` property has been removed.

Reason for change

Automatically compacting the cache caused problems. To avoid unexpected behavior, the cache should only be compacted when needed.

Recommended action

To compact the cache, downcast to `MemoryCache` and call `Compact` when needed.

Category

ASP.NET Core

Affected APIs

[MemoryCacheOptions.CompactOnMemoryPressure](#)

Caching: Microsoft.Extensions.Caching.SqlServer uses new SqlClient package

The `Microsoft.Extensions.Caching.SqlServer` package will use the new `Microsoft.Data.SqlClient` package instead of `System.Data.SqlClient` package. This change could cause slight behavioral breaking changes. For more information, see [Introducing the new Microsoft.Data.SqlClient](#).

Version introduced

3.0

Old behavior

The `Microsoft.Extensions.Caching.SqlServer` package used the `System.Data.SqlClient` package.

New behavior

`Microsoft.Extensions.Caching.SqlServer` is now using the `Microsoft.Data.SqlClient` package.

Reason for change

`Microsoft.Data.SqlClient` is a new package that is built off of `System.Data.SqlClient`. It's where all new feature work will be done from now on.

Recommended action

Customers shouldn't need to worry about this breaking change unless they were using types returned by the `Microsoft.Extensions.Caching.SqlServer` package and casting them to `System.Data.SqlClient` types. For example, if someone was casting a `DbConnection` to the [old `SqlConnection` type](#), they would need to change the cast to the new `Microsoft.Data.SqlClient.SqlConnection` type.

Category

ASP.NET Core

Affected APIs

None

Caching: ResponseCaching "pubternal" types changed to internal

In ASP.NET Core 3.0, "pubternal" types in `ResponseCaching` have been changed to `internal`.

In addition, default implementations of `IResponseCachingPolicyProvider` and `IResponseCachingKeyProvider` are no longer added to services as part of the `AddResponseCaching` method.

Change description

In ASP.NET Core, "pubternal" types are declared as `public` but reside in a namespace suffixed with `.Internal`. While these types are public, they have no support policy and are subject to breaking changes. Unfortunately, accidental use of these types has been common, resulting in breaking changes to these projects and limiting the ability to maintain the framework.

Version introduced

3.0

Old behavior

These types were publicly visible, but unsupported.

New behavior

These types are now `internal`.

Reason for change

The `internal` scope better reflects the unsupported policy.

Recommended action

Copy types that are used by your app or library.

Category

ASP.NET Core

Affected APIs

- `Microsoft.AspNetCore.ResponseCaching.Internal.CachedResponse`
- `Microsoft.AspNetCore.ResponseCaching.Internal.CachedVaryByRules`
- `Microsoft.AspNetCore.ResponseCaching.Internal.IResponseCache`
- `Microsoft.AspNetCore.ResponseCaching.Internal.IResponseCacheEntry`
- `Microsoft.AspNetCore.ResponseCaching.Internal.IResponseCachingKeyProvider`
- `Microsoft.AspNetCore.ResponseCaching.Internal.IResponseCachingPolicyProvider`
- `Microsoft.AspNetCore.ResponseCaching.Internal.MemoryResponseCache`
- `Microsoft.AspNetCore.ResponseCaching.Internal.ResponseCachingContext`
- `Microsoft.AspNetCore.ResponseCaching.Internal.ResponseCachingKeyProvider`
- `Microsoft.AspNetCore.ResponseCaching.Internal.ResponseCachingPolicyProvider`
- `Microsoft.AspNetCore.ResponseCaching.ResponseCachingMiddleware.ResponseCachingMiddleware(RequestDelegate, IOptions<ResponseCachingOptions>, ILoggerFactory, IResponseCachingPolicyProvider, IResponseCache, IResponseCachingKeyProvider)`

Data Protection: `DataProtection.Blobs` uses new Azure Storage APIs

`Azure.Extensions.AspNetCore.DataProtection.Blobs` depends on the [Azure Storage libraries](#). These libraries renamed their assemblies, packages, and namespaces. Starting in ASP.NET Core 3.0, `Azure.Extensions.AspNetCore.DataProtection.Blobs` uses the new `Azure.Storage.`-prefixed APIs and packages.

For questions about the Azure Storage APIs, use <https://github.com/Azure/azure-storage-net>. For discussion on this issue, see [dotnet/aspnetcore#19570](https://github.com/dotnet/aspnetcore/issues/19570).

Version introduced

3.0

Old behavior

The package referenced the `WindowsAzure.Storage` NuGet package. The package references the `Microsoft.Azure.Storage.Blob` NuGet package.

New behavior

The package references the `Azure.Storage.Blob` NuGet package.

Reason for change

This change allows `Azure.Extensions.AspNetCore.DataProtection.Blobs` to migrate to the recommended Azure Storage packages.

Recommended action

If you still need to use the older Azure Storage APIs with ASP.NET Core 3.0, add a direct dependency to the package [WindowsAzure.Storage](#) or [Microsoft.Azure.Storage](#). This package can be installed alongside the new `Azure.Storage` APIs.

In many cases, the upgrade only involves changing the `using` statements to use the new namespaces:

diff

```
- using Microsoft.WindowsAzure.Storage;
- using Microsoft.WindowsAzure.Storage.Blob;
- using Microsoft.Azure.Storage;
- using Microsoft.Azure.Storage.Blob;
```

```
+ using Azure.Storage;  
+ using Azure.Storage.Blobs;
```

Category

ASP.NET Core

Affected APIs

None

Hosting: AspNetCoreModule V1 removed from Windows Hosting Bundle

Starting with ASP.NET Core 3.0, the Windows Hosting Bundle won't contain AspNetCoreModule (ANCM) V1.

ANCM V2 is backwards compatible with ANCM OutOfProcess and is recommended for use with ASP.NET Core 3.0 apps.

For discussion, see [dotnet/aspnetcore#7095](#).

Version introduced

3.0

Old behavior

ANCM V1 is included in the Windows Hosting Bundle.

New behavior

ANCM V1 isn't included in the Windows Hosting Bundle.

Reason for change

ANCM V2 is backwards compatible with ANCM OutOfProcess and is recommended for use with ASP.NET Core 3.0 apps.

Recommended action

Use ANCM V2 with ASP.NET Core 3.0 apps.

If ANCM V1 is required, it can be installed using the ASP.NET Core 2.1 or 2.2 Windows Hosting Bundle.

This change will break ASP.NET Core 3.0 apps that:

- Explicitly opted into using ANCM V1 with
`<AspNetCoreModuleName>AspNetCoreModule</AspNetCoreModuleName>`.
- Have a custom `web.config` file with `<add name="aspNetCore" path="*" verb="*" modules="AspNetCoreModule" resourceType="Unspecified" />`.

Category

ASP.NET Core

Affected APIs

None

Hosting: Generic host restricts Startup constructor injection

The only types the generic host supports for `Startup` class constructor injection are `IHostEnvironment`, `IWebHostEnvironment`, and `IConfiguration`. Apps using `WebHost` are unaffected.

Change description

Prior to ASP.NET Core 3.0, constructor injection could be used for arbitrary types in the `Startup` class's constructor. In ASP.NET Core 3.0, the web stack was replatformed onto the generic host library. You can see the change in the `Program.cs` file of the templates:

ASP.NET Core 2.x:

[https://github.com/dotnet/aspnetcore/blob/5cb615fcbe8559e49042e93394008077e30454c0/src/Templating/src/Microsoft.DotNet.Web.ProjectTemplates/content/EmptyWeb-CSharp/Program.cs#L20-L22 ↗](https://github.com/dotnet/aspnetcore/blob/5cb615fcbe8559e49042e93394008077e30454c0/src/Templating/src/Microsoft.DotNet.Web.ProjectTemplates/content/EmptyWeb-CSharp/Program.cs#L20-L22)

ASP.NET Core 3.0:

[https://github.com/dotnet/aspnetcore/blob/b1ca2c1155da3920f0df5108b9fedbe82efaa11c/src/ProjectTemplates/Web.ProjectTemplates/content/EmptyWeb-CSharp/Program.cs#L19-L24 ↗](https://github.com/dotnet/aspnetcore/blob/b1ca2c1155da3920f0df5108b9fedbe82efaa11c/src/ProjectTemplates/Web.ProjectTemplates/content/EmptyWeb-CSharp/Program.cs#L19-L24)

`Host` uses one dependency injection (DI) container to build the app. `WebHost` uses two containers: one for the host and one for the app. As a result, the `Startup` constructor no longer supports custom service injection. Only `IHostEnvironment`, `IWebHostEnvironment`, and `IConfiguration` can be injected. This change prevents DI issues such as the duplicate creation of a singleton service.

Version introduced

3.0

Reason for change

This change is a consequence of replatforming the web stack onto the generic host library.

Recommended action

Inject services into the `Startup.Configure` method signature. For example:

C#

```
public void Configure(IApplicationBuilder app, IOptions<MyOptions> options)
```

Category

ASP.NET Core

Affected APIs

None

Hosting: HTTPS redirection enabled for IIS out-of-process apps

Version 13.0.19218.0 of the [ASP.NET Core Module \(ANCM\)](#) for hosting via IIS out-of-process enables an existing HTTPS redirection feature for ASP.NET Core 3.0 and 2.2

apps.

For discussion, see [dotnet/AspNetCore#15243](#).

Version introduced

3.0

Old behavior

The ASP.NET Core 2.1 project template first introduced support for HTTPS middleware methods like [UseHttpsRedirection](#) and [UseHsts](#). Enabling HTTPS redirection required the addition of configuration, since apps in development don't use the default port of 443. [HTTP Strict Transport Security \(HSTS\)](#) is active only if the request is already using HTTPS. Localhost is skipped by default.

New behavior

In ASP.NET Core 3.0, the IIS HTTPS scenario was [enhanced](#). With the enhancement, an app could discover the server's HTTPS ports and make [UseHttpsRedirection](#) work by default. The in-process component accomplished port discovery with the [IServerAddresses](#) feature, which only affects ASP.NET Core 3.0 apps because the in-process library is versioned with the framework. The out-of-process component changed to automatically add the [ASPNETCORE_HTTPS_PORT](#) environment variable. This change affected both ASP.NET Core 2.2 and 3.0 apps because the out-of-process component is shared globally. ASP.NET Core 2.1 apps aren't affected because they use a prior version of ANCM by default.

The preceding behavior was modified in ASP.NET Core 3.0.1 and 3.1.0 Preview 3 to reverse the behavior changes in ASP.NET Core 2.x. These changes only affect IIS out-of-process apps.

As detailed above, installing ASP.NET Core 3.0.0 had the side effect of also activating the [UseHttpsRedirection](#) middleware in ASP.NET Core 2.x apps. A change was made to ANCM in ASP.NET Core 3.0.1 and 3.1.0 Preview 3 such that installing them no longer has this effect on ASP.NET Core 2.x apps. The [ASPNETCORE_HTTPS_PORT](#) environment variable that ANCM populated in ASP.NET Core 3.0.0 was changed to [ASPNETCORE_ANCM_HTTPS_PORT](#) in ASP.NET Core 3.0.1 and 3.1.0 Preview 3.

[UseHttpsRedirection](#) was also updated in these releases to understand both the new and old variables. ASP.NET Core 2.x won't be updated. As a result, it reverts to the previous behavior of being disabled by default.

Reason for change

Improved ASP.NET Core 3.0 functionality.

Recommended action

No action is required if you want all clients to use HTTPS. To allow some clients to use HTTP, take one of the following steps:

- Remove the calls to `UseHttpsRedirection` and `UseHsts` from your project's `Startup.Configure` method, and redeploy the app.
- In your `web.config` file, set the `ASPNETCORE_HTTPS_PORT` environment variable to an empty string. This change can occur directly on the server without redeploying the app. For example:

XML

```
<aspNetCore processPath="dotnet" arguments=".\\WebApplication3.dll"
stdoutEnabled="false" stdoutLogFile="\\?\%home%\LogFiles\stdout" >
<environmentVariables>
<environmentVariable name="ASPNETCORE_HTTPS_PORT" value="" />
</environmentVariables>
</aspNetCore>
```

`UseHttpsRedirection` can still be:

- Activated manually in ASP.NET Core 2.x by setting the `ASPNETCORE_HTTPS_PORT` environment variable to the appropriate port number (443 in most production scenarios).
- Deactivated in ASP.NET Core 3.x by defining `ASPNETCORE_ANCM_HTTPS_PORT` with an empty string value. This value is set in the same fashion as the preceding `ASPNETCORE_HTTPS_PORT` example.

Machines running ASP.NET Core 3.0.0 apps should install the ASP.NET Core 3.0.1 runtime before installing the ASP.NET Core 3.1.0 Preview 3 ANCM. Doing so ensures that `UseHttpsRedirection` continues to operate as expected for the ASP.NET Core 3.0 apps.

In Azure App Service, ANCM deploys on a separate schedule from the runtime because of its global nature. ANCM was deployed to Azure with these changes after ASP.NET Core 3.0.1 and 3.1.0 were deployed.

Category

ASP.NET Core

Affected APIs

[HttpsPolicyBuilderExtensions.UseHttpsRedirection\(IApplicationBuilder\)](#)

Hosting: `IHostingEnvironment` and `IApplicationLifetime` types marked obsolete and replaced

New types have been introduced to replace existing `IHostingEnvironment` and `IApplicationLifetime` types.

Version introduced

3.0

Old behavior

There were two different `IHostingEnvironment` and `IApplicationLifetime` types from `Microsoft.Extensions.Hosting` and `Microsoft.AspNetCore.Hosting`.

New behavior

The old types have been marked as obsolete and replaced with new types.

Reason for change

When `Microsoft.Extensions.Hosting` was introduced in ASP.NET Core 2.1, some types like `IHostingEnvironment` and `IApplicationLifetime` were copied from `Microsoft.AspNetCore.Hosting`. Some ASP.NET Core 3.0 changes cause apps to include both the `Microsoft.Extensions.Hosting` and `Microsoft.AspNetCore.Hosting` namespaces. Any use of those duplicate types causes an "ambiguous reference" compiler error when both namespaces are referenced.

Recommended action

Replaced any usages of the old types with the newly introduced types as below:

Obsolete types (warning):

- [Microsoft.Extensions.Hosting.IHostingEnvironment](#)
- [Microsoft.AspNetCore.Hosting.IHostingEnvironment](#)
- [Microsoft.Extensions.Hosting.IApplicationLifetime](#)
- [Microsoft.AspNetCore.Hosting.IApplicationLifetime](#)
- [Microsoft.Extensions.Hosting.EnvironmentName](#)
- [Microsoft.AspNetCore.Hosting.EnvironmentName](#)

New types:

- [Microsoft.Extensions.Hosting.IHostEnvironment](#)
- [Microsoft.AspNetCore.Hosting.IWebHostEnvironment : IHostEnvironment](#)
- [Microsoft.Extensions.Hosting.IHostApplicationLifetime](#)
- [Microsoft.Extensions.Hosting.Environments](#)

The new `IHostEnvironment`, `IsDevelopment` and `IsProduction` extension methods are in the `Microsoft.Extensions.Hosting` namespace. That namespace may need to be added to your project.

Category

ASP.NET Core

Affected APIs

- [Microsoft.AspNetCore.Hosting.EnvironmentName](#)
- [Microsoft.AspNetCore.Hosting.IApplicationLifetime](#)
- [Microsoft.AspNetCore.Hosting.IHostingEnvironment](#)
- [Microsoft.Extensions.Hosting.EnvironmentName](#)
- [Microsoft.Extensions.Hosting.IApplicationLifetime](#)
- [Microsoft.Extensions.Hosting.IHostingEnvironment](#)

Hosting: ObjectPoolProvider removed from WebHostBuilder dependencies

As part of making ASP.NET Core more pay for play, the `ObjectPoolProvider` was removed from the main set of dependencies. Specific components relying on `ObjectPoolProvider` now add it themselves.

For discussion, see [dotnet/aspnetcore#5944](#).

Version introduced

3.0

Old behavior

`WebHostBuilder` provides `ObjectPoolProvider` by default in the DI container.

New behavior

`WebHostBuilder` no longer provides `ObjectPoolProvider` by default in the DI container.

Reason for change

This change was made to make ASP.NET Core more pay for play.

Recommended action

If your component requires `ObjectPoolProvider`, it needs to be added to your dependencies via the `IServiceCollection`.

Category

ASP.NET Core

Affected APIs

None

HTTP: DefaultHttpContext extensibility removed

As part of ASP.NET Core 3.0 performance improvements, the extensibility of `DefaultHttpContext` was removed. The class is now `sealed`. For more information, see [dotnet/aspnetcore#6504](#).

If your unit tests use `Mock<DefaultHttpContext>`, use `Mock<HttpContext>` or `new DefaultHttpContext()` instead.

For discussion, see [dotnet/aspnetcore#6534](#).

Version introduced

3.0

Old behavior

Classes can derive from `DefaultHttpContext`.

New behavior

Classes can't derive from `DefaultHttpContext`.

Reason for change

The extensibility was provided initially to allow pooling of the `HttpContext`, but it introduced unnecessary complexity and impeded other optimizations.

Recommended action

If you're using `Mock<DefaultHttpContext>` in your unit tests, begin using `Mock<HttpContext>` instead.

Category

ASP.NET Core

Affected APIs

[Microsoft.AspNetCore.Http.DefaultHttpContext](#)

HTTP: HeaderNames constants changed to static readonly

Starting in ASP.NET Core 3.0 Preview 5, the fields in `Microsoft.Net.Http.Headers.HeaderNames` changed from `const` to `static readonly`.

For discussion, see [dotnet/aspnetcore#9514](#).

Version introduced

3.0

Old behavior

These fields used to be `const`.

New behavior

These fields are now `static readonly`.

Reason for change

The change:

- Prevents the values from being embedded across assembly boundaries, allowing for value corrections as needed.
- Enables faster reference equality checks.

Recommended action

Recompile against 3.0. Source code using these fields in the following ways can no longer do so:

- As an attribute argument
- As a `case` in a `switch` statement
- When defining another `const`

To work around the breaking change, switch to using self-defined header name constants or string literals.

Category

ASP.NET Core

Affected APIs

[Microsoft.Net.Http.Headers.HeaderNames](#)

HTTP: Response body infrastructure changes

The infrastructure backing an HTTP response body has changed. If you're using `HttpResponse` directly, you shouldn't need to make any code changes. Read further if you're wrapping or replacing `HttpResponse.Body` or accessing `HttpContext.Features`.

Version introduced

3.0

Old behavior

There were three APIs associated with the HTTP response body:

- `IHttpResponseFeature.Body`
- `IHttpSendFileFeature.SendFileAsync`
- `IHttpBufferingFeature.DisableResponseBuffering`

New behavior

If you replace `HttpResponse.Body`, it replaces the entire `IHttpResponseBodyFeature` with a wrapper around your given stream using `StreamResponseBodyFeature` to provide default implementations for all of the expected APIs. Setting back the original stream reverts this change.

Reason for change

The motivation is to combine the response body APIs into a single new feature interface.

Recommended action

Use `IHttpResponseBodyFeature` where you previously were using `IHttpResponseFeature.Body`, `IHttpSendFileFeature`, or `IHttpBufferingFeature`.

Category

ASP.NET Core

Affected APIs

- [Microsoft.AspNetCore.Http.Features.IHttpBufferingFeature](#)
- [Microsoft.AspNetCore.Http.Features.IHttpResponseFeature.Body](#)

- Microsoft.AspNetCore.Http.Features.IHttpSendFileFeature
-

HTTP: Some cookie SameSite defaults changed to None

`SameSite` is an option for cookies that can help mitigate some Cross-Site Request Forgery (CSRF) attacks. When this option was initially introduced, inconsistent defaults were used across various ASP.NET Core APIs. The inconsistency has led to confusing results. As of ASP.NET Core 3.0, these defaults are better aligned. You must opt in to this feature on a per-component basis.

Version introduced

3.0

Old behavior

Similar ASP.NET Core APIs used different default `SameSiteMode` values. An example of the inconsistency is seen in `HttpResponse.Cookies.Append(String, String)` and `HttpResponse.Cookies.Append(String, String, CookieOptions)`, which defaulted to `SameSiteMode.None` and `SameSiteMode.Lax`, respectively.

New behavior

All the affected APIs default to `SameSiteMode.None`.

Reason for change

The default value was changed to make `SameSite` an opt-in feature.

Recommended action

Each component that emits cookies needs to decide if `SameSite` is appropriate for its scenarios. Review your usage of the affected APIs and reconfigure `SameSite` as needed.

Category

ASP.NET Core

Affected APIs

- `IResponseCookies.Append(String, String, CookieOptions)`
 - `CookiePolicyOptions.MinimumSameSitePolicy`
-

HTTP: Synchronous IO disabled in all servers

Starting with ASP.NET Core 3.0, synchronous server operations are disabled by default.

Change description

`AllowSynchronousIO` is an option in each server that enables or disables synchronous IO APIs like `HttpRequest.Body.Read`, `HttpResponse.Body.Write`, and `Stream.Flush`. These APIs have long been a source of thread starvation and app hangs. Starting in ASP.NET Core 3.0 Preview 3, these synchronous operations are disabled by default.

Affected servers:

- Kestrel
- HttpSys
- IIS in-process
- TestServer

Expect errors similar to:

- Synchronous operations are disallowed. Call `ReadAsync` or set `AllowSynchronousIO` to `true` instead.
- Synchronous operations are disallowed. Call `WriteAsync` or set `AllowSynchronousIO` to `true` instead.
- Synchronous operations are disallowed. Call `FlushAsync` or set `AllowSynchronousIO` to `true` instead.

Each server has an `AllowSynchronousIO` option that controls this behavior and the default for all of them is now `false`.

The behavior can also be overridden on a per-request basis as a temporary mitigation. For example:

C#

```
var syncIOFeature = HttpContext.Features.Get<IHttpBodyControlFeature>();
if (syncIOFeature != null)
{
```

```
    syncIOFeature.AllowSynchronousIO = true;  
}
```

If you have trouble with a `TextWriter` or another stream calling a synchronous API in `Dispose`, call the new `DisposeAsync` API instead.

For discussion, see [dotnet/aspnetcore#7644](#).

Version introduced

3.0

Old behavior

`HttpRequest.Body.Read`, `HttpResponse.Body.Write`, and `Stream.Flush` were allowed by default.

New behavior

These synchronous APIs are disallowed by default:

Expect errors similar to:

- Synchronous operations are disallowed. Call `ReadAsync` or set `AllowSynchronousIO` to `true` instead.
- Synchronous operations are disallowed. Call `WriteAsync` or set `AllowSynchronousIO` to `true` instead.
- Synchronous operations are disallowed. Call `FlushAsync` or set `AllowSynchronousIO` to `true` instead.

Reason for change

These synchronous APIs have long been a source of thread starvation and app hangs. Starting in ASP.NET Core 3.0 Preview 3, the synchronous operations are disabled by default.

Recommended action

Use the asynchronous versions of the methods. The behavior can also be overridden on a per-request basis as a temporary mitigation.

C#

```
var syncIOFeature = HttpContext.Features.Get<IHttpBodyControlFeature>();  
if (syncIOFeature != null)  
{  
    syncIOFeature.AllowSynchronousIO = true;  
}
```

Category

ASP.NET Core

Affected APIs

- [Stream.Flush](#)
 - [Stream.Read](#)
 - [Stream.Write](#)
-

Identity: AddDefaultUI method overload removed

Starting with ASP.NET Core 3.0, the [IdentityBuilderUIExtensions.AddDefaultUI\(IdentityBuilder, UIFramework\)](#) method overload no longer exists.

Version introduced

3.0

Reason for change

This change was a result of adoption of the static web assets feature.

Recommended action

Call [IdentityBuilderUIExtensions.AddDefaultUI\(IdentityBuilder\)](#) instead of the overload that takes two arguments. If you're using Bootstrap 3, also add the following line to a `<PropertyGroup>` element in your project file:

XML

```
<IdentityUIFrameworkVersion>Bootstrap3</IdentityUIFrameworkVersion>
```

Category

ASP.NET Core

Affected APIs

[IdentityBuilderUIExtensions.AddDefaultUI\(IdentityBuilder, UIFramework\)](#)

Identity: Default Bootstrap version of UI changed

Starting in ASP.NET Core 3.0, Identity UI defaults to using version 4 of Bootstrap.

Version introduced

3.0

Old behavior

The `services.AddDefaultIdentity<IdentityUser>().AddDefaultUI();` method call was the same as `services.AddDefaultIdentity<IdentityUser>().AddDefaultUI(UIFramework.Bootstrap3);`

New behavior

The `services.AddDefaultIdentity<IdentityUser>().AddDefaultUI();` method call is the same as `services.AddDefaultIdentity<IdentityUser>().AddDefaultUI(UIFramework.Bootstrap4);`

Reason for change

Bootstrap 4 was released during ASP.NET Core 3.0 timeframe.

Recommended action

You're impacted by this change if you use the default Identity UI and have added it in `Startup.ConfigureServices` as shown in the following example:

C#

```
services.AddDefaultIdentity<IdentityUser>().AddDefaultUI();
```

Take one of the following actions:

- Migrate your app to use Bootstrap 4 using their [migration guide ↗](#).
- Update `Startup.ConfigureServices` to enforce usage of Bootstrap 3. For example:

C#

```
services.AddDefaultIdentity<IdentityUser>
    ().AddDefaultUI(UIFramework.Bootstrap3);
```

Category

ASP.NET Core

Affected APIs

None

Identity: SignInAsync throws exception for unauthenticated identity

By default, `SignInAsync` throws an exception for principals / identities in which `IsAuthenticated` is `false`.

Version introduced

3.0

Old behavior

`SignInAsync` accepts any principals / identities, including identities in which `IsAuthenticated` is `false`.

New behavior

By default, `SignInAsync` throws an exception for principals / identities in which `IsAuthenticated` is `false`. There's a new flag to suppress this behavior, but the default behavior has changed.

Reason for change

The old behavior was problematic because, by default, these principals were rejected by `[Authorize]` / `RequireAuthenticatedUser()`.

Recommended action

In ASP.NET Core 3.0 Preview 6, there's a `RequireAuthenticatedSignIn` flag on `AuthenticationOptions` that is `true` by default. Set this flag to `false` to restore the old behavior.

Category

ASP.NET Core

Affected APIs

None

Identity: SignInManager constructor accepts new parameter

Starting with ASP.NET Core 3.0, a new `IUserConfirmation<TUser>` parameter was added to the `SignInManager` constructor. For more information, see [dotnet/aspnetcore#8356](#).

Version introduced

3.0

Reason for change

The motivation for the change was to add support for new email / confirmation flows in Identity.

Recommended action

If manually constructing a `SignInManager`, provide an implementation of `IUserConfirmation` or grab one from dependency injection to provide.

Category

ASP.NET Core

Affected APIs

[SignInManager<TUser>](#)

Identity: UI uses static web assets feature

ASP.NET Core 3.0 introduced a static web assets feature, and Identity UI has adopted it.

Change description

As a result of Identity UI adopting the static web assets feature:

- Framework selection is accomplished by using the `IdentityUIFrameworkVersion` property in your project file.
- Bootstrap 4 is the default UI framework for Identity UI. Bootstrap 3 has reached end of life, and you should consider migrating to a supported version.

Version introduced

3.0

Old behavior

The default UI framework for Identity UI was **Bootstrap 3**. The UI framework could be configured using a parameter to the `AddDefaultUI` method call in

`Startup.ConfigureServices`.

New behavior

The default UI framework for Identity UI is **Bootstrap 4**. The UI framework must be configured in your project file, instead of in the `AddDefaultUI` method call.

Reason for change

Adoption of the static web assets feature required that the UI framework configuration move to MSBuild. The decision on which framework to embed is a build-time decision, not a runtime decision.

Recommended action

Review your site UI to ensure the new Bootstrap 4 components are compatible. If necessary, use the `IdentityUIFrameworkVersion` MSBuild property to revert to Bootstrap 3. Add the property to a `<PropertyGroup>` element in your project file:

XML

```
<IdentityUIFrameworkVersion>Bootstrap3</IdentityUIFrameworkVersion>
```

Category

ASP.NET Core

Affected APIs

[IdentityBuilderUIExtensions.AddDefaultUI\(IdentityBuilder, UIFramework\)](#)

Kestrel: Connection adapters removed

As part of the move to move "pubternal" APIs to `public`, the concept of an `IConnectionAdapter` was removed from Kestrel. Connection adapters are being replaced with connection middleware (similar to HTTP middleware in the ASP.NET Core pipeline, but for lower-level connections). HTTPS and connection logging have moved from connection adapters to connection middleware. Those extension methods should continue to work seamlessly, but the implementation details have changed.

For more information, see [dotnet/aspnetcore#11412](#). For discussion, see [dotnet/aspnetcore#11475](#).

Version introduced

3.0

Old behavior

Kestrel extensibility components were created using `IConnectionAdapter`.

New behavior

Kestrel extensibility components are created as [middleware](#).

Reason for change

This change is intended to provide a more flexible extensibility architecture.

Recommended action

Convert any implementations of `IConnectionAdapter` to use the new middleware pattern as shown [here](#).

Category

ASP.NET Core

Affected APIs

`Microsoft.AspNetCore.Server.Kestrel.Core.Adapter.Internal.IConnectionAdapter`

Kestrel: Empty HTTPS assembly removed

The assembly [Microsoft.AspNetCore.Server.Kestrel.Https](#) has been removed.

Version introduced

3.0

Reason for change

In ASP.NET Core 2.1, the contents of `Microsoft.AspNetCore.Server.Kestrel.Https` were moved to [Microsoft.AspNetCore.Server.Kestrel.Core](#). This change was done in a non-breaking way using `[TypeForwardedTo]` attributes.

Recommended action

- Libraries referencing `Microsoft.AspNetCore.Server.Kestrel.Https` 2.0 should update all ASP.NET Core dependencies to 2.1 or later. Otherwise, they may break when loaded into an ASP.NET Core 3.0 app.
- Apps and libraries targeting ASP.NET Core 2.1 and later should remove any direct references to the `Microsoft.AspNetCore.Server.Kestrel.Https` NuGet package.

Category

ASP.NET Core

Affected APIs

None

Kestrel: Request trailer headers moved to new collection

In prior versions, Kestrel added HTTP/1.1 chunked trailer headers into the request headers collection when the request body was read to the end. This behavior caused concerns about ambiguity between headers and trailers. The decision was made to move the trailers to a new collection.

HTTP/2 request trailers were unavailable in ASP.NET Core 2.2 but are now also available in this new collection in ASP.NET Core 3.0.

New request extension methods have been added to access these trailers.

HTTP/1.1 trailers are available once the entire request body has been read.

HTTP/2 trailers are available once they're received from the client. The client won't send the trailers until the entire request body has been at least buffered by the server. You may need to read the request body to free up buffer space. Trailers are always available if you read the request body to the end. The trailers mark the end of the body.

Version introduced

3.0

Old behavior

Request trailer headers would be added to the `HttpRequest.Headers` collection.

New behavior

Request trailer headers aren't present in the `HttpRequest.Headers` collection. Use the following extension methods on `HttpRequest` to access them:

- `GetDeclaredTrailers()` - Gets the request "Trailer" header that lists which trailers to expect after the body.
- `SupportsTrailers()` - Indicates if the request supports receiving trailer headers.
- `CheckTrailersAvailable()` - Determines if the request supports trailers and if they're available for reading.
- `GetTrailer(string trailerName)` - Gets the requested trailing header from the response.

Reason for change

Trailers are a key feature in scenarios like gRPC. Merging the trailers in to request headers was confusing to users.

Recommended action

Use the trailer-related extension methods on `HttpRequest` to access trailers.

Category

ASP.NET Core

Affected APIs

[HttpRequest.Headers](#)

Kestrel: Transport abstractions removed and made public

As part of moving away from "pubternal" APIs, the Kestrel transport layer APIs are exposed as a public interface in the `Microsoft.AspNetCore.Connections.Abstractions` library.

Version introduced

3.0

Old behavior

- Transport-related abstractions were available in the `Microsoft.AspNetCore.Server.Kestrel.Transport.Abstractions` library.
- The `ListenOptions.NoDelay` property was available.

New behavior

- The `IConnectionListener` interface was introduced in the `Microsoft.AspNetCore.Connections.Abstractions` library to expose the most used functionality from the `...Transport.Abstractions` library.
- The `NoDelay` is now available in transport options (`LibuvTransportOptions` and `SocketTransportOptions`).
- `SchedulingMode` is no longer available.

Reason for change

ASP.NET Core 3.0 has moved away from "pubternal" APIs.

Recommended action

Category

ASP.NET Core

Affected APIs

None

Localization: `ResourceManagerWithCultureStringLocalizer` and `WithCulture` marked obsolete

The `ResourceManagerWithCultureStringLocalizer` class and `WithCulture` interface member are often sources of confusion for users of localization, especially when creating their own `IStringLocalizer` implementation. These items give the user the impression that an `IStringLocalizer` instance is "per-language, per-resource". In reality, the instances should only be "per-resource". The language searched for is determined by the `CultureInfo.CurrentCulture` at execution time. To eliminate the source of

confusion, the APIs were marked as obsolete in ASP.NET Core 3.0 Preview 3. The APIs will be removed in a future release.

For context, see [dotnet/aspnetcore#3324](#). For discussion, see [dotnet/aspnetcore#7756](#).

Version introduced

3.0

Old behavior

Methods weren't marked as `Obsolete`.

New behavior

Methods are marked `Obsolete`.

Reason for change

The APIs represented a use case that isn't recommended. There was confusion about the design of localization.

Recommended action

The recommendation is to use `ResourceManagerStringLocalizer` instead. Let the culture be set by the `CurrentCulture`. If that isn't an option, create and use a copy of [ResourceManagerWithCultureStringLocalizer](#).

Category

ASP.NET Core

Affected APIs

- [ResourceManagerWithCultureStringLocalizer](#)
- [ResourceManagerStringLocalizer.WithCulture](#)

Logging: DebugLogger class made internal

Prior to ASP.NET Core 3.0, `DebugLogger`'s access modifier was `public`. In ASP.NET Core 3.0, the access modifier changed to `internal`.

Version introduced

3.0

Reason for change

The change is being made to:

- Enforce consistency with other logger implementations such as `ConsoleLogger`.
- Reduce the API surface.

Recommended action

Use the [AddDebug `ILoggingBuilder`](#) extension method to enable debug logging.

`DebugLoggerProvider` is also still `public` in the event the service needs to be registered manually.

Category

ASP.NET Core

Affected APIs

[Microsoft.Extensions.Logging.Debug.DebugLogger](#)

MVC: Async suffix trimmed from controller action names

As part of addressing [dotnet/aspnetcore#4849](#), ASP.NET Core MVC trims the suffix `Async` from action names by default. Starting with ASP.NET Core 3.0, this change affects both routing and link generation.

Version introduced

3.0

Old behavior

Consider the following ASP.NET Core MVC controller:

C#

```
public class ProductController : Controller
{
    public async IActionResult ListAsync()
    {
        var model = await DbContext.Products.ToListAsync();
        return View(model);
    }
}
```

The action is routable via `Product/ListAsync`. Link generation requires specifying the `Async` suffix. For example:

CSHTML

```
<a asp-controller="Product" asp-action="ListAsync">List</a>
```

New behavior

In ASP.NET Core 3.0, the action is routable via `Product/List`. Link generation code should omit the `Async` suffix. For example:

CSHTML

```
<a asp-controller="Product" asp-action="List">List</a>
```

This change doesn't affect names specified using the `[ActionName]` attribute. The new behavior can be disabled by setting `MvcOptions.SuppressAsyncSuffixInActionNames` to `false` in `Startup.ConfigureServices`:

C#

```
services.AddMvc(options =>
{
    options.SuppressAsyncSuffixInActionNames = false;
});
```

Reason for change

By convention, asynchronous .NET methods are suffixed with `Async`. However, when a method defines an MVC action, it's undesirable to use the `Async` suffix.

Recommended action

If your app depends on MVC actions preserving the name's `Async` suffix, choose one of the following mitigations:

- Use the `[ActionName]` attribute to preserve the original name.
- Disable the renaming entirely by setting

```
MvcOptions.SuppressAsyncSuffixInActionNames to false in  
Startup.ConfigureServices:
```

```
C#
```

```
services.AddMvc(options =>  
{  
    options.SuppressAsyncSuffixInActionNames = false;  
});
```

Category

ASP.NET Core

Affected APIs

None

MVC: JsonResult moved to Microsoft.AspNetCore.Mvc.Core

`JsonResult` has moved to the `Microsoft.AspNetCore.Mvc.Core` assembly. This type used to be defined in [Microsoft.AspNetCore.Mvc.Formatters.Json](#). An assembly-level `[TypeForwardedTo]` attribute was added to `Microsoft.AspNetCore.Mvc.Formatters.Json` to address this issue for the majority of users. Apps that use third-party libraries may encounter issues.

Version introduced

3.0 Preview 6

Old behavior

An app using a 2.2-based library builds successfully.

New behavior

An app using a 2.2-based library fails compilation. An error containing a variation of the following text is provided:

Output

```
The type ' JsonResult ' exists in both ' Microsoft.AspNetCore.Mvc.Core , Version=3.0.0.0, Culture=neutral, PublicKeyToken=adb9793829ddae60 ' and  
' Microsoft.AspNetCore.Mvc.Formatters.Json , Version=2.0.0.0, Culture=neutral, PublicKeyToken=adb9793829ddae60 '
```

For an example of such an issue, see [dotnet/aspnetcore#7220](#).

Reason for change

Platform-level changes to the composition of ASP.NET Core as described at [aspnet/Announcements#325](#).

Recommended action

Libraries compiled against the 2.2 version of `Microsoft.AspNetCore.Mvc.Formatters.Json` may need to recompile to address the problem for all consumers. If affected, contact the library author. Request recompilation of the library to target ASP.NET Core 3.0.

Category

ASP.NET Core

Affected APIs

[Microsoft.AspNetCore.Mvc.JsonResult](#)

MVC: Precompilation tool deprecated

In ASP.NET Core 1.1, the `Microsoft.AspNetCore.Mvc.Razor.ViewCompilation` (MVC precompilation tool) package was introduced to add support for publish-time

compilation of Razor files (.cshtml files). In ASP.NET Core 2.1, the [Razor SDK](#) was introduced to expand upon features of the precompilation tool. The Razor SDK added support for build- and publish-time compilation of Razor files. The SDK verifies the correctness of .cshtml files at build time while improving on app startup time. The Razor SDK is on by default, and no gesture is required to start using it.

In ASP.NET Core 3.0, the ASP.NET Core 1.1-era MVC precompilation tool was removed. Earlier package versions will continue receiving important bug and security fixes in the patch release.

Version introduced

3.0

Old behavior

The `Microsoft.AspNetCore.Mvc.Razor.ViewCompilation` package was used to pre-compile MVC Razor views.

New behavior

The Razor SDK natively supports this functionality. The `Microsoft.AspNetCore.Mvc.Razor.ViewCompilation` package is no longer updated.

Reason for change

The Razor SDK provides more functionality and verifies the correctness of .cshtml files at build time. The SDK also improves app startup time.

Recommended action

For users of ASP.NET Core 2.1 or later, update to use the native support for precompilation in the [Razor SDK](#). If bugs or missing features prevent migration to the Razor SDK, open an issue at [dotnet/aspnetcore](#).

Category

ASP.NET Core

Affected APIs

None

MVC: "Pubternal" types changed to internal

In ASP.NET Core 3.0, all "pubternal" types in MVC were updated to either be `public` in a supported namespace or `internal` as appropriate.

Change description

In ASP.NET Core, "pubternal" types are declared as `public` but reside in a `.Internal`-suffixed namespace. While these types are `public`, they have no support policy and are subject to breaking changes. Unfortunately, accidental use of these types has been common, resulting in breaking changes to these projects and limiting the ability to maintain the framework.

Version introduced

3.0

Old behavior

Some types in MVC were `public` but in a `.Internal` namespace. These types had no support policy and were subject to breaking changes.

New behavior

All such types are updated either to be `public` in a supported namespace or marked as `internal`.

Reason for change

Accidental use of the "pubternal" types has been common, resulting in breaking changes to these projects and limiting the ability to maintain the framework.

Recommended action

If you're using types that have become truly `public` and have been moved into a new, supported namespace, update your references to match the new namespaces.

If you're using types that have become marked as `internal`, you'll need to find an alternative. The previously "pubternal" types were never supported for public use. If there are specific types in these namespaces that are critical to your apps, file an issue at [dotnet/aspnetcore](#). Considerations may be made for making the requested types `public`.

Category

ASP.NET Core

Affected APIs

This change includes types in the following namespaces:

- `Microsoft.AspNetCore.Mvc.Cors.Internal`
 - `Microsoft.AspNetCore.Mvc.DataAnnotations.Internal`
 - `Microsoft.AspNetCore.Mvc.Formatters.Internal`
 - `Microsoft.AspNetCore.Mvc.Formatters.Json.Internal`
 - `Microsoft.AspNetCore.Mvc.Formatters.Xml.Internal`
 - `Microsoft.AspNetCore.Mvc.Internal`
 - `Microsoft.AspNetCore.Mvc.ModelBinding.Internal`
 - `Microsoft.AspNetCore.Mvc.Razor.Internal`
 - `Microsoft.AspNetCore.Mvc.RazorPages.Internal`
 - `Microsoft.AspNetCore.Mvc.TagHelpers.Internal`
 - `Microsoft.AspNetCore.Mvc.ViewFeatures.Internal`
-

MVC: Web API compatibility shim removed

Starting with ASP.NET Core 3.0, the `Microsoft.AspNetCore.Mvc.WebApiCompatShim` package is no longer available.

Change description

The `Microsoft.AspNetCore.Mvc.WebApiCompatShim` (`WebApiCompatShim`) package provides partial compatibility in ASP.NET Core with ASP.NET 4.x Web API 2 to simplify migrating existing Web API implementations to ASP.NET Core. However, apps using the `WebApiCompatShim` don't benefit from the API-related features shipping in recent ASP.NET Core releases. Such features include improved Open API specification generation, standardized error handling, and client code generation. To better focus the

API efforts in 3.0, `WebApiCompatShim` was removed. Existing apps using the `WebApiCompatShim` should migrate to the newer `[ApiController]` model.

Version introduced

3.0

Reason for change

The Web API compatibility shim was a migration tool. It restricts user access to new functionality added in ASP.NET Core.

Recommended action

Remove usage of this shim and migrate directly to the similar functionality in ASP.NET Core itself.

Category

ASP.NET Core

Affected APIs

[Microsoft.AspNetCore.Mvc.WebApiCompatShim](#)

Razor: RazorTemplateEngine API removed

The `RazorTemplateEngine` API was removed and replaced with `Microsoft.AspNetCore.Razor.Language.RazorProjectEngine`.

For discussion, see GitHub issue [dotnet/aspnetcore#25215](#).

Version introduced

3.0

Old behavior

A template engine can be created and used to parse and generate code for Razor files.

New behavior

`RazorProjectEngine` can be created and provided the same type of information as `RazorTemplateEngine` to parse and generate code for Razor files. `RazorProjectEngine` also provides extra levels of configuration.

Reason for change

`RazorTemplateEngine` was too tightly coupled to the existing implementations. This tight coupling led to more questions when trying to properly configure a Razor parsing/generation pipeline.

Recommended action

Use `RazorProjectEngine` instead of `RazorTemplateEngine`. Consider the following examples.

Create and configure the RazorProjectEngine

C#

```
RazorProjectEngine projectEngine =
    RazorProjectEngine.Create(RazorConfiguration.Default,

RazorProjectFileSystem.Create(@"C:\source\repos\ConsoleApp4\ConsoleApp4"),
    builder =>
{
    builder.ConfigureClass((document, classNode) =>
    {
        classNode.ClassName = "MyClassName";

        // Can also configure other aspects of the class here.
    });

    // More configuration can go here
});
```

Generate code for a Razor file

C#

```
RazorProjectItem item = projectEngine.FileSystem.GetItem(
    @"C:\source\repos\ConsoleApp4\ConsoleApp4\Example.cshtml",
    FileKinds.Legacy);
RazorCodeDocument output = projectEngine.Process(item);
```

```
// Things available
RazorSyntaxTree syntaxTree = output.GetSyntaxTree();
DocumentIntermediateNode intermediateDocument =
    output.GetDocumentIntermediateNode();
RazorCSharpDocument csharpDocument = output.GetCSharpDocument();
```

Category

ASP.NET Core

Affected APIs

- `RazorTemplateEngine`
 - `RazorTemplateEngineOptions`
-

Razor: Runtime compilation moved to a package

Support for runtime compilation of Razor views and Razor Pages has moved to a separate package.

Version introduced

3.0

Old behavior

Runtime compilation is available without needing additional packages.

New behavior

The functionality has been moved to the [Microsoft.AspNetCore.Mvc.Razor.RuntimeCompilation](#) package.

The following APIs were previously available in

`Microsoft.AspNetCore.Mvc.Razor.RazorViewEngineOptions` to support runtime compilation. The APIs are now available via `Microsoft.AspNetCore.Mvc.Razor.RuntimeCompilation.MvcRazorRuntimeCompilationOption`s.

- `RazorViewEngineOptions.FileProviders` is now `MvcRazorRuntimeCompilationOptions.FileProviders`
- `RazorViewEngineOptions.AdditionalCompilationReferences` is now `MvcRazorRuntimeCompilationOptions.AdditionalReferencePaths`

In addition,

`Microsoft.AspNetCore.Mvc.Razor.RazorViewEngineOptions.AllowRecompilingViewsOnFileChange` has been removed. Recompilation on file changes is enabled by default by referencing the `Microsoft.AspNetCore.Mvc.Razor.RuntimeCompilation` package.

Reason for change

This change was necessary to remove the ASP.NET Core shared framework dependency on Roslyn.

Recommended action

Apps that require runtime compilation or recompilation of Razor files should take the following steps:

1. Add a reference to the `Microsoft.AspNetCore.Mvc.Razor.RuntimeCompilation` package.
2. Update the project's `Startup.ConfigureServices` method to include a call to `AddRazorRuntimeCompilation`. For example:

C#

```
services.AddMvc()
    .AddRazorRuntimeCompilation();
```

Category

ASP.NET Core

Affected APIs

[Microsoft.AspNetCore.Mvc.Razor.RazorViewEngineOptions](#)

Session state: Obsolete APIs removed

Obsolete APIs for configuring session cookies were removed. For more information, see [aspnet/Announcements#257](#).

Version introduced

3.0

Reason for change

This change enforces consistency across APIs for configuring features that use cookies.

Recommended action

Migrate usage of the removed APIs to their newer replacements. Consider the following example in `Startup.ConfigureServices`:

C#

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddSession(options =>
    {
        // Removed obsolete APIs
        options.CookieName = "SessionCookie";
        options.CookieDomain = "contoso.com";
        options.CookiePath = "/";
        options.CookieHttpOnly = true;
        options.CookieSecure = CookieSecurePolicy.Always;

        // new API
        options.Cookie.Name = "SessionCookie";
        options.Cookie.Domain = "contoso.com";
        options.Cookie.Path = "/";
        options.Cookie.HttpOnly = true;
        options.Cookie.SecurePolicy = CookieSecurePolicy.Always;
    });
}
```

Category

ASP.NET Core

Affected APIs

- `Microsoft.AspNetCore.Builder.SessionOptions.CookieDomain`

- Microsoft.AspNetCore.Builder.SessionOptions.CookieHttpOnly
 - Microsoft.AspNetCore.Builder.SessionOptions.CookieName
 - Microsoft.AspNetCore.Builder.SessionOptions.CookiePath
 - Microsoft.AspNetCore.Builder.SessionOptions.CookieSecure
-

Shared framework: Assemblies removed from Microsoft.AspNetCore.App

Starting in ASP.NET Core 3.0, the ASP.NET Core shared framework (`Microsoft.AspNetCore.App`) only contains first-party assemblies that are fully developed, supported, and serviceable by Microsoft.

Change description

Think of the change as the redefining of boundaries for the ASP.NET Core "platform." The shared framework will be [source-buildable by anybody via GitHub](#) and will continue to offer the existing benefits of .NET Core shared frameworks to your apps. Some benefits include smaller deployment size, centralized patching, and faster startup time.

As part of the change, some notable breaking changes are introduced in `Microsoft.AspNetCore.App`.

Version introduced

3.0

Old behavior

Projects referenced `Microsoft.AspNetCore.App` via a `<PackageReference>` element in the project file.

Additionally, `Microsoft.AspNetCore.App` contained the following subcomponents:

- Json.NET (`Newtonsoft.Json`)
- Entity Framework Core (assemblies prefixed with `Microsoft.EntityFrameworkCore.`)
- Roslyn (`Microsoft.CodeAnalysis`)

New behavior

A reference to `Microsoft.AspNetCore.App` no longer requires a `<PackageReference>` element in the project file. The .NET Core SDK supports a new element called `<FrameworkReference>`, which replaces the use of `<PackageReference>`.

For more information, see [dotnet/aspnetcore#3612](#).

Entity Framework Core ships as NuGet packages. This change aligns the shipping model with all other data access libraries on .NET. It provides Entity Framework Core the simplest path to continue innovating while supporting the various .NET platforms. The move of Entity Framework Core out of the shared framework has no impact on its status as a Microsoft-developed, supported, and serviceable library. The [.NET Core support policy](#) continues to cover it.

Json.NET and Entity Framework Core continue to work with ASP.NET Core. They won't, however, be included in the shared framework.

For more information, see [The future of JSON in .NET Core 3.0](#). Also see [the complete list of binaries](#) removed from the shared framework.

Reason for change

This change simplifies the consumption of `Microsoft.AspNetCore.App` and reduces the duplication between NuGet packages and shared frameworks.

For more information on the motivation for this change, see [this blog post](#).

Recommended action

Starting with ASP.NET Core 3.0, it is no longer necessary for projects to consume assemblies in `Microsoft.AspNetCore.App` as NuGet packages. To simplify the targeting and usage of the ASP.NET Core shared framework, many NuGet packages shipped since ASP.NET Core 1.0 are no longer produced. The APIs those packages provide are still available to apps by using a `<FrameworkReference>` to `Microsoft.AspNetCore.App`. Common API examples include Kestrel, MVC, and Razor.

This change doesn't apply to all binaries referenced via `Microsoft.AspNetCore.App` in ASP.NET Core 2.x. Notable exceptions include:

- `Microsoft.Extensions` libraries that continue to target .NET Standard are available as NuGet packages (see <https://github.com/dotnet/extensions>).
- APIs produced by the ASP.NET Core team that aren't part of `Microsoft.AspNetCore.App`. For example, the following components are available as

NuGet packages:

- Entity Framework Core
- APIs that provide third-party integration
- Experimental features
- APIs with dependencies that couldn't [fulfill the requirements to be in the shared framework](#)
- Extensions to MVC that maintain support for Json.NET. An API is provided as [a NuGet package](#) to support using Json.NET and MVC. See the [ASP.NET Core migration guide for more details](#).
- The SignalR .NET client continues to support .NET Standard and ships as [a NuGet package](#). It's intended for use on many .NET runtimes, such as Xamarin and UWP.

For more information, see [Stop producing packages for shared framework assemblies in 3.0](#). For discussion, see [dotnet/aspnetcore#3757](#).

Category

ASP.NET Core

Affected APIs

- [Microsoft.CodeAnalysis](#)
- [Microsoft.EntityFrameworkCore](#)

Shared framework: Removed Microsoft.AspNetCore.All

Starting in ASP.NET Core 3.0, the `Microsoft.AspNetCore.All` metapackage and the matching `Microsoft.AspNetCore.All` shared framework are no longer produced. This package is available in ASP.NET Core 2.2 and will continue to receive servicing updates in ASP.NET Core 2.1.

Version introduced

3.0

Old behavior

Apps could use the `Microsoft.AspNetCore.All` metapackage to target the `Microsoft.AspNetCore.All` shared framework on .NET Core.

New behavior

.NET Core 3.0 doesn't include a `Microsoft.AspNetCore.All` shared framework.

Reason for change

The `Microsoft.AspNetCore.All` metapackage included a large number of external dependencies.

Recommended action

Migrate your project to use the `Microsoft.AspNetCore.App` framework. Components that were previously available in `Microsoft.AspNetCore.All` are still available on NuGet. Those components are now deployed with your app instead of being included in the shared framework.

Category

ASP.NET Core

Affected APIs

None

SignalR: `HandshakeProtocol.SuccessHandshakeData` replaced

The `HandshakeProtocol.SuccessHandshakeData` field was removed and replaced with a helper method that generates a successful handshake response given a specific `IHubProtocol`.

Version introduced

3.0

Old behavior

`HandshakeProtocol.SuccessHandshakeData` was a `public static ReadOnlyMemory<byte>` field.

New behavior

`HandshakeProtocol.SuccessHandshakeData` has been replaced by a `static GetSuccessfulHandshake(IHubProtocol protocol)` method that returns a `ReadOnlyMemory<byte>` based on the specified protocol.

Reason for change

Additional fields were added to the handshake *response* that are non-constant and change depending on the selected protocol.

Recommended action

None. This type isn't designed for use from user code. It's `public`, so it can be shared between the SignalR server and client. It may also be used by customer SignalR clients written in .NET. **Users** of SignalR shouldn't be affected by this change.

Category

ASP.NET Core

Affected APIs

[HandshakeProtocol.SuccessHandshakeData](#)

SignalR: HubConnection ResetSendPing and ResetTimeout methods removed

The `ResetSendPing` and `ResetTimeout` methods were removed from the SignalR `HubConnection` API. These methods were originally intended only for internal use but were made public in ASP.NET Core 2.2. These methods won't be available starting in the ASP.NET Core 3.0 Preview 4 release. For discussion, see [dotnet/aspnetcore#8543](#).

Version introduced

3.0

Old behavior

APIs were available.

New behavior

APIs are removed.

Reason for change

These methods were originally intended only for internal use but were made public in ASP.NET Core 2.2.

Recommended action

Don't use these methods.

Category

ASP.NET Core

Affected APIs

- [HubConnection.ResetSendPing\(\)](#)
 - [HubConnection.ResetTimeout\(\)](#)
-

SignalR: HubConnectionContext constructors changed

SignalR's `HubConnectionContext` constructors changed to accept an options type, rather than multiple parameters, to future-proof adding options. This change replaces two constructors with a single constructor that accepts an options type.

Version introduced

3.0

Old behavior

`HubConnectionContext` has two constructors:

C#

```
public HubConnectionContext(ConnectionString connectionContext, TimeSpan  
keepAliveInterval, ILoggerFactory loggerFactory);  
public HubConnectionContext(ConnectionString connectionContext, TimeSpan
```

```
keepAliveInterval, ILoggerFactory loggerFactory, TimeSpan  
clientTimeoutInterval);
```

New behavior

The two constructors were removed and replaced with one constructor:

C#

```
public HubConnectionContext(ConnectionString connectionContext,  
HubConnectionContextOptions contextOptions, ILoggerFactory loggerFactory)
```

Reason for change

The new constructor uses a new options object. Consequently, the features of `HubConnectionContext` can be expanded in the future without making more constructors and breaking changes.

Recommended action

Instead of using the following constructor:

C#

```
HubConnectionContext connectionContext = new HubConnectionContext(  
    connectionContext,  
    keepAliveInterval: TimeSpan.FromSeconds(15),  
    loggerFactory,  
    clientTimeoutInterval: TimeSpan.FromSeconds(15));
```

Use the following constructor:

C#

```
HubConnectionContextOptions contextOptions = new  
HubConnectionContextOptions()  
{  
    KeepAliveInterval = TimeSpan.FromSeconds(15),  
    ClientTimeoutInterval = TimeSpan.FromSeconds(15)  
};  
HubConnectionContext connectionContext = new  
HubConnectionContext(connectionContext, contextOptions, loggerFactory);
```

Category

ASP.NET Core

Affected APIs

- `HubConnectionContext(ConnectionString, TimeSpan, ILoggerFactory)`
 - `HubConnectionContext(ConnectionString, TimeSpan, ILoggerFactory, TimeSpan)`
-

SignalR: JavaScript client package name changed

In ASP.NET Core 3.0 Preview 7, the SignalR JavaScript client package name changed from `@aspnet/signalr` to `@microsoft/signalr`. The name change reflects the fact that SignalR is useful in more than just ASP.NET Core apps, thanks to the Azure SignalR Service.

To react to this change, change references in your `package.json` files, `require` statements, and ECMAScript `import` statements. No API will change as part of this rename.

For discussion, see [dotnet/aspnetcore#11637](#).

Version introduced

3.0

Old behavior

The client package was named `@aspnet/signalr`.

New behavior

The client package is named `@microsoft/signalr`.

Reason for change

The name change clarifies that SignalR is useful beyond ASP.NET Core apps, thanks to the Azure SignalR Service.

Recommended action

Switch to the new package `@microsoft/signalr`.

Category

ASP.NET Core

Affected APIs

None

SignalR: `UseSignalR` and `UseConnections` methods marked obsolete

The methods `UseConnections` and `UseSignalR` and the classes `ConnectionsRouteBuilder` and `HubRouteBuilder` are marked as obsolete in ASP.NET Core 3.0.

Version introduced

3.0

Old behavior

SignalR hub routing was configured using `UseSignalR` or `UseConnections`.

New behavior

The old way of configuring routing has been obsoleted and replaced with endpoint routing.

Reason for change

Middleware is being moved to the new endpoint routing system. The old way of adding middleware is being obsoleted.

Recommended action

Replace `UseSignalR` with `UseEndpoints`:

Old code:

C#

```
app.UseSignalR(routes =>
{
    routes.MapHub<SomeHub>("/path");
});
```

New code:

C#

```
app.UseEndpoints(endpoints =>
{
    endpoints.MapHub<SomeHub>("/path");
});
```

Category

ASP.NET Core

Affected APIs

- [Microsoft.AspNetCore.Builder.ConnectionsAppBuilderExtensions.UseConnections\(IApplicationBuilder, Action<ConnectionsRouteBuilder>\)](#)
- [Microsoft.AspNetCore.Builder.SignalRAppBuilderExtensions.UseSignalR\(IApplicationBuilder, Action<HubRouteBuilder>\)](#)
- [Microsoft.AspNetCore.Http.Connections.ConnectionsRouteBuilder](#)
- [Microsoft.AspNetCore.SignalR.HubRouteBuilder](#)

SPAs: SpaServices and NodeServices marked obsolete

The contents of the following NuGet packages have all been unnecessary since ASP.NET Core 2.1. Consequently, the following packages are being marked as obsolete:

- [Microsoft.AspNetCore.SpaServices](#)
- [Microsoft.AspNetCore.NodeServices](#)

For the same reason, the following npm modules are being marked as deprecated:

- [aspnet-angular](#)
- [aspnet-prerendering](#)
- [aspnet-webpack](#)

- [aspnet-webpack-react](#)
- [domain-task](#)

The preceding packages and npm modules will later be removed in .NET 5.

Version introduced

3.0

Old behavior

The deprecated packages and npm modules were intended to integrate ASP.NET Core with various Single-Page App (SPA) frameworks. Such frameworks include Angular, React, and React with Redux.

New behavior

A new integration mechanism exists in the [Microsoft.AspNetCore.SpaServices.Extensions](#) NuGet package. The package remains the basis of the Angular and React project templates since ASP.NET Core 2.1.

Reason for change

ASP.NET Core supports integration with various Single-Page App (SPA) frameworks, including Angular, React, and React with Redux. Initially, integration with these frameworks was accomplished with ASP.NET Core-specific components that handled scenarios like server-side prerendering and integration with Webpack. As time went on, industry standards changed. Each of the SPA frameworks released their own standard command-line interfaces. For example, Angular CLI and create-react-app.

When ASP.NET Core 2.1 was released in May 2018, the team responded to the change in standards. A newer and simpler way to integrate with the SPA frameworks' own toolchains was provided. The new integration mechanism exists in the package [Microsoft.AspNetCore.SpaServices.Extensions](#) and remains the basis of the Angular and React project templates since ASP.NET Core 2.1.

To clarify that the older ASP.NET Core-specific components are irrelevant and not recommended:

- The pre-2.1 integration mechanism is marked as obsolete.
- The supporting npm packages are marked as deprecated.

Recommended action

If you're using these packages, update your apps to use the functionality:

- In the `Microsoft.AspNetCore.SpaServices.Extensions` package.
- Provided by the SPA frameworks you're using

To enable features like server-side prerendering and hot module reload, see the documentation for the corresponding SPA framework. The functionality in `Microsoft.AspNetCore.SpaServices.Extensions` is *not* obsolete and will continue to be supported.

Category

ASP.NET Core

Affected APIs

- [Microsoft.AspNetCore.Builder.SpaRouteExtensions](#)
- [Microsoft.AspNetCore.Builder.WebpackDevMiddleware](#)
- [Microsoft.AspNetCore.NodeServices.EmbeddedResourceReader](#)
- [Microsoft.AspNetCore.NodeServices.INodeServices](#)
- [Microsoft.AspNetCore.NodeServices.NodeServicesFactory](#)
- [Microsoft.AspNetCore.NodeServices.NodeServicesOptions](#)
- [Microsoft.AspNetCore.NodeServices.StringAsTempFile](#)
- [Microsoft.AspNetCore.NodeServices.HostingModels.INodeInstance](#)
- [Microsoft.AspNetCore.NodeServices.HostingModels.NodeInvocationException](#)
- [Microsoft.AspNetCore.NodeServices.HostingModels.NodeInvocationInfo](#)
- [Microsoft.AspNetCore.NodeServices.HostingModels.NodeServicesOptionsExtensions](#)
- [Microsoft.AspNetCore.NodeServices.HostingModels.OutOfProcessNodeInstance](#)
- [Microsoft.AspNetCore.SpaServices.Prerendering.ISpaPrerenderer](#)
- [Microsoft.AspNetCore.SpaServices.Prerendering.ISpaPrerendererBuilder](#)

- [Microsoft.AspNetCore.SpaServices.Prerendering.JavaScriptModuleExport](#)
 - [Microsoft.AspNetCore.SpaServices.Prerendering.Prerenderer](#)
 - [Microsoft.AspNetCore.SpaServices.Prerendering.PrerenderTagHelper](#)
 - [Microsoft.AspNetCore.SpaServices.Prerendering.RenderToStringResult](#)
 - [Microsoft.AspNetCore.SpaServices.Webpack.WebpackDevMiddlewareOptions](#)
 - [Microsoft.Extensions.DependencyInjection.NodeServicesServiceCollectionExtensions](#)
 - [Microsoft.Extensions.DependencyInjection.PrerenderingServiceCollectionExtensions](#)
-

SPAs: `SpaServices` and `NodeServices` no longer fall back to console logger

`Microsoft.AspNetCore.SpaServices` and `Microsoft.AspNetCore.NodeServices` won't display console logs unless logging is configured.

Version introduced

3.0

Old behavior

`Microsoft.AspNetCore.SpaServices` and `Microsoft.AspNetCore.NodeServices` used to automatically create a console logger when logging isn't configured.

New behavior

`Microsoft.AspNetCore.SpaServices` and `Microsoft.AspNetCore.NodeServices` won't display console logs unless logging is configured.

Reason for change

There's a need to align with how other ASP.NET Core packages implement logging.

Recommended action

If the old behavior is required, to configure console logging, add `services.AddLogging(builder => builder.AddConsole())` to your `Setup.ConfigureServices` method.

Category

ASP.NET Core

Affected APIs

None

Target framework: .NET Framework support dropped

Starting with ASP.NET Core 3.0, .NET Framework is an unsupported target framework.

Change description

.NET Framework 4.8 is the last major version of .NET Framework. New ASP.NET Core apps should be built on .NET Core. Starting with the .NET Core 3.0 release, you can think of ASP.NET Core 3.0 as being part of .NET Core.

Customers using ASP.NET Core with .NET Framework can continue in a fully supported fashion using the [2.1 LTS release](#). Support and servicing for 2.1 continues until at least August 21, 2021. This date is three years after declaration of the LTS release per the [.NET Support Policy](#). Support for ASP.NET Core 2.1 packages **on .NET Framework** will extend indefinitely, similar to the [servicing policy for other package-based ASP.NET frameworks](#).

For more information about porting from .NET Framework to .NET Core, see [Porting to .NET Core](#).

`Microsoft.Extensions` packages (such as logging, dependency injection, and configuration) and Entity Framework Core aren't affected. They'll continue to support .NET Standard.

For more information on the motivation for this change, see [the original blog post](#).

Version introduced

3.0

Old behavior

ASP.NET Core apps could run on either .NET Core or .NET Framework.

New behavior

ASP.NET Core apps can only be run on .NET Core.

Recommended action

Take one of the following actions:

- Keep your app on ASP.NET Core 2.1.
- Migrate your app and dependencies to .NET Core.

Category

ASP.NET Core

Affected APIs

None

Core .NET libraries

- APIs that report version now report product and not file version
- Custom EncoderFallbackBuffer instances cannot fall back recursively
- Floating point formatting and parsing behavior changes
- Floating-point parsing operations no longer fail or throw an OverflowException
- InvalidAsynchronousStateException moved to another assembly
- Replacing ill-formed UTF-8 byte sequences follows Unicode guidelines
- TypeDescriptionProviderAttribute moved to another assembly
- ZipArchiveEntry no longer handles archives with inconsistent entry sizes
- FieldInfo.SetValue throws exception for static, init-only fields
- Passing GroupCollection to extension methods taking `IEnumerable<T>` requires disambiguation

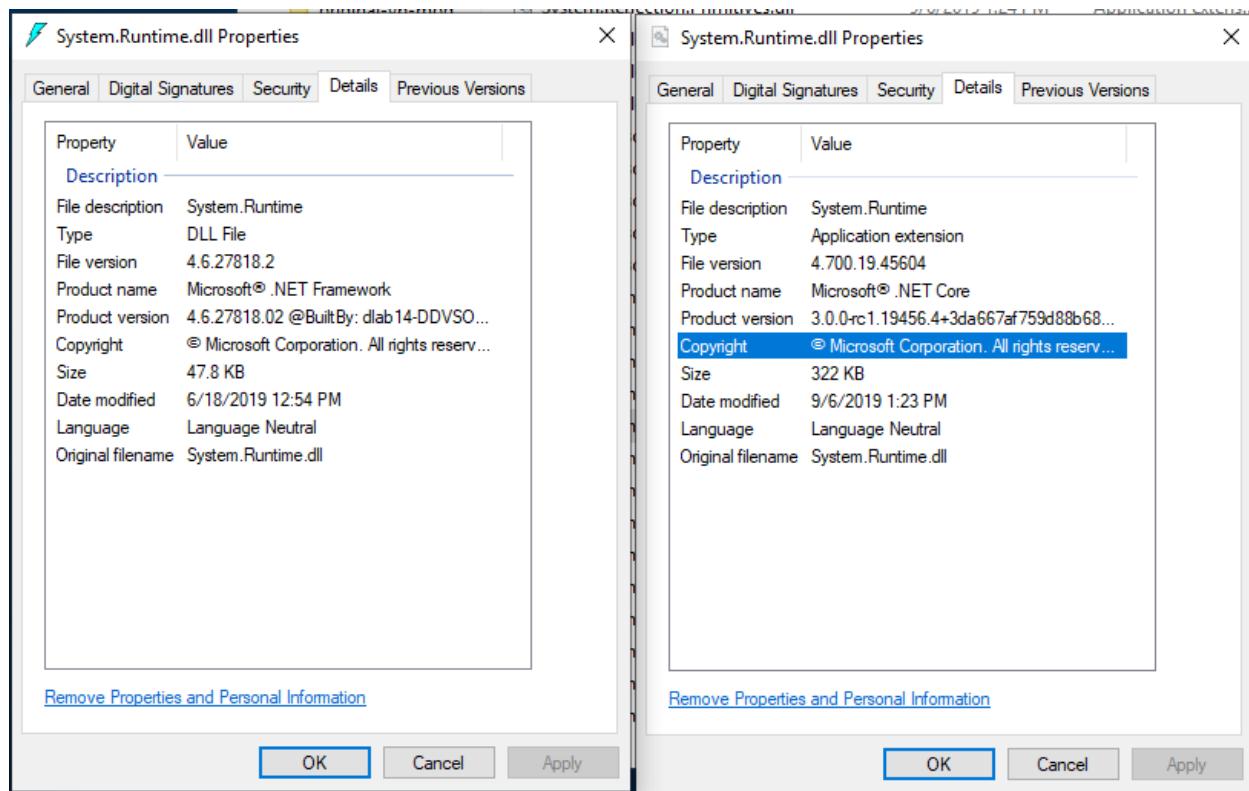
APIs that report version now report product and not file version

Many of the APIs that return versions in .NET Core now return the product version rather than the file version.

Change description

In .NET Core 2.2 and previous versions, methods such as [Environment.Version](#), [RuntimeInformation.FrameworkDescription](#), and the file properties dialog for .NET Core assemblies reflect the file version. Starting with .NET Core 3.0, they reflect the product version.

The following figure illustrates the difference in version information for the *System.Runtime.dll* assembly for .NET Core 2.2 (on the left) and .NET Core 3.0 (on the right) as displayed by the **Windows Explorer** file properties dialog.



Version introduced

3.0

Recommended action

None. This change should make version detection intuitive rather than obtuse.

Category

Affected APIs

- [Environment.Version](#)
 - [RuntimeInformation.FrameworkDescription](#)
-

Custom EncoderFallbackBuffer instances cannot fall back recursively

Custom [EncoderFallbackBuffer](#) instances cannot fall back recursively. The implementation of [EncoderFallbackBuffer.GetNextChar\(\)](#) must result in a character sequence that is convertible to the destination encoding. Otherwise, an exception occurs.

Change description

During a character-to-byte transcoding operation, the runtime detects ill-formed or nonconvertible UTF-16 sequences and provides those characters to the [EncoderFallbackBuffer.Fallback](#) method. The `Fallback` method determines which characters should be substituted for the original nonconvertible data, and these characters are drained by calling [EncoderFallbackBuffer.GetNextChar](#) in a loop.

The runtime then attempts to transcode these substitution characters to the target encoding. If this operation succeeds, the runtime continues transcoding from where it left off in the original input string.

Previously, custom implementations of [EncoderFallbackBuffer.GetNextChar\(\)](#) can return character sequences that are not convertible to the destination encoding. If the substituted characters cannot be transcoded to the target encoding, the runtime invokes the [EncoderFallbackBuffer.Fallback](#) method once again with the substitution characters, expecting the [EncoderFallbackBuffer.GetNextChar\(\)](#) method to return a new substitution sequence. This process continues until the runtime eventually sees a well-formed, convertible substitution, or until a maximum recursion count is reached.

Starting with .NET Core 3.0, custom implementations of [EncoderFallbackBuffer.GetNextChar\(\)](#) must return character sequences that are convertible to the destination encoding. If the substituted characters cannot be transcoded to the target encoding, an [ArgumentException](#) is thrown. The runtime will no longer make recursive calls into the [EncoderFallbackBuffer](#) instance.

This behavior only applies when all three of the following conditions are met:

- The runtime detects an ill-formed UTF-16 sequence or a UTF-16 sequence that cannot be converted to the target encoding.
- A custom [EncoderFallback](#) has been specified.
- The custom [EncoderFallback](#) attempts to substitute a new ill-formed or nonconvertible UTF-16 sequence.

Version introduced

3.0

Recommended action

Most developers needn't take any action.

If an application uses a custom [EncoderFallback](#) and [EncoderFallbackBuffer](#) class, ensure the implementation of [EncoderFallbackBuffer.Fallback](#) populates the fallback buffer with well-formed UTF-16 data that is directly convertible to the target encoding when the [Fallback](#) method is first invoked by the runtime.

Category

Core .NET libraries

Affected APIs

- [EncoderFallbackBuffer.Fallback](#)
- [EncoderFallbackBuffer.GetNextChar\(\)](#)

Floating-point formatting and parsing behavior changed

Floating-point parsing and formatting behavior (by the [Double](#) and [Single](#) types) are now [IEEE-compliant](#). This ensures that the behavior of floating-point types in .NET matches that of other IEEE-compliant languages. For example,

`double.Parse("SomeLiteral")` should always match what C# produces for `double x = SomeLiteral.`

Change description

In .NET Core 2.2 and earlier versions, formatting with `Double.ToString` and `Single.ToString`, and parsing with `Double.Parse`, `Double.TryParse`, `Single.Parse`, and `Single.TryParse` are not IEEE-compliant. As a result, it's impossible to guarantee that a value will roundtrip with any supported standard or custom format string. For some inputs, the attempt to parse a formatted value can fail, and for others, the parsed value doesn't equal the original value.

Starting with .NET Core 3.0, floating-point parsing and formatting operations are IEEE 754-compliant.

The following table shows two code snippets and how the output changes between .NET Core 2.2 and .NET Core 3.1.

| Code snippet | Output on .NET Core 2.2 | Output on .NET Core 3.1 |
|---|-------------------------|-------------------------|
| <pre>Console.WriteLine((-0.0).ToString());</pre> | 0 | -0 |
| <pre>var value = -3.123456789123456789; Console.WriteLine(value == double.Parse(value.ToString()));</pre> | False | True |

For more information, see the [Floating-point parsing and formatting improvements in .NET Core 3.0](#) blog post.

Version introduced

3.0

Recommended action

The [Potential impact to existing code](#) section of the [Floating-point parsing and formatting improvements in .NET Core 3.0](#) blog post suggests some changes you can make to your code if you want to maintain the previous behavior.

- For some differences in formatting, you can get behavior equivalent to the previous behavior by specifying a different format string.
- For differences in parsing, there's no mechanism to fall back to the previous behavior.

Category

Core .NET libraries

Affected APIs

- [Double.ToString](#)
 - [Single.ToString](#)
 - [Double.Parse](#)
 - [Double.TryParse](#)
 - [Single.Parse](#)
 - [Single.TryParse](#)
-

Floating-point parsing operations no longer fail or throw an `OverflowException`

The floating-point parsing methods no longer throw an [OverflowException](#) or return `false` when they parse a string whose numeric value is outside the range of the [Single](#) or [Double](#) floating-point type.

Change description

In .NET Core 2.2 and earlier versions, the [Double.Parse](#) and [Single.Parse](#) methods throw an [OverflowException](#) for values that outside the range of their respective type. The [Double.TryParse](#) and [Single.TryParse](#) methods return `false` for the string representations of out-of-range numeric values.

Starting with .NET Core 3.0, the [Double.Parse](#), [Double.TryParse](#), [Single.Parse](#), and [Single.TryParse](#) methods no longer fail when parsing out-of-range numeric strings. Instead, the [Double](#) parsing methods return [Double.PositiveInfinity](#) for values that exceed [Double.MaxValue](#), and they return [Double.NegativeInfinity](#) for values that are less than [Double.MinValue](#). Similarly, the [Single](#) parsing methods return [Single.PositiveInfinity](#) for values that exceed [Single.MaxValue](#), and they return [Single.NegativeInfinity](#) for values that are less than [Single.MinValue](#).

This change was made for improved IEEE 754:2008 compliance.

Version introduced

3.0

Recommended action

This change can affect your code in either of two ways:

- Your code depends on the handler for the `OverflowException` to execute when an overflow occurs. In this case, you should remove the `catch` statement and place any necessary code in an `If` statement that tests whether `Double.IsInfinity` or `Single.IsInfinity` is `true`.
- Your code assumes that floating-point values are not `Infinity`. In this case, you should add the necessary code to check for floating-point values of `PositiveInfinity` and `NegativeInfinity`.

Category

Core .NET libraries

Affected APIs

- `Double.Parse`
- `Double.TryParse`
- `Single.Parse`
- `Single.TryParse`

InvalidAsynchronousStateException moved to another assembly

The `InvalidAsynchronousStateException` class has been moved.

Change description

In .NET Core 2.2 and earlier versions, the `InvalidAsynchronousStateException` class is found in the `System.ComponentModel.TypeConverter` assembly.

Starting with .NET Core 3.0, it is found in the `System.ComponentModel.Primitives` assembly.

Version introduced

3.0

Recommended action

This change only affects applications that use reflection to load the [InvalidOperationException](#) by calling a method such as [Assembly.GetType](#) or an overload of [Activator.CreateInstance](#) that assumes the type is in a particular assembly. If that is the case, update the assembly referenced in the method call to reflect the type's new assembly location.

Category

Core .NET libraries

Affected APIs

None.

Replacing ill-formed UTF-8 byte sequences follows Unicode guidelines

When the [UTF8Encoding](#) class encounters an ill-formed UTF-8 byte sequence during a byte-to-character transcoding operation, it replaces that sequence with a '߿' (U+FFFD REPLACEMENT CHARACTER) character in the output string. .NET Core 3.0 differs from previous versions of .NET Core and the .NET Framework by following the Unicode best practice for performing this replacement during the transcoding operation.

This is part of a larger effort to improve UTF-8 handling throughout .NET, including by the new [System.Text.Unicode.Utf8](#) and [System.Text.Rune](#) types. The [UTF8Encoding](#) type was given improved error handling mechanics so that it produces output consistent with the newly introduced types.

Change description

Starting with .NET Core 3.0, when transcoding bytes to characters, the [UTF8Encoding](#) class performs character substitution based on Unicode best practices. The substitution mechanism used is described by [The Unicode Standard, Version 12.0, Sec. 3.9 \(PDF\)](#) in the heading titled *U+FFFD Substitution of Maximal Subparts*.

This behavior *only* applies when the input byte sequence contains ill-formed UTF-8 data. Additionally, if the [UTF8Encoding](#) instance has been constructed with `throwOnInvalidBytes: true`, the [UTF8Encoding](#) instance will continue to throw on invalid input rather than perform U+FFFD replacement. For more information about the [UTF8Encoding](#) constructor, see [UTF8Encoding\(Boolean, Boolean\)](#).

The following table illustrates the impact of this change with an invalid 3-byte input:

| III-formed 3-byte input | Output before .NET Core 3.0 | Output starting with .NET Core 3.0 |
|-------------------------|------------------------------------|---|
| [ED A0 90] | [FFFD FFFD] (2-character output) | [FFFD FFFD FFFD] (3-character output) |

The 3-char output is the preferred output, according to *Table 3-9* of the previously linked Unicode Standard PDF.

Version introduced

3.0

Recommended action

No action is required on the part of the developer.

Category

Core .NET libraries

Affected APIs

- [UTF8Encoding.GetCharCount](#)
- [UTF8Encoding.GetChars](#)
- [UTF8Encoding.GetString\(Byte\[\], Int32, Int32\)](#)

TypeDescriptionProviderAttribute moved to another assembly

The [TypeDescriptionProviderAttribute](#) class has been moved.

Change description

In .NET Core 2.2 and earlier versions, The [TypeDescriptionProviderAttribute](#) class is found in the *System.ComponentModel.TypeConverter* assembly.

Starting with .NET Core 3.0, it is found in the *System.ObjectModel* assembly.

Version introduced

3.0

Recommended action

This change only affects applications that use reflection to load the [TypeDescriptionProviderAttribute](#) type by calling a method such as [Assembly.GetType](#) or an overload of [Activator.CreateInstance](#) that assumes the type is in a particular assembly. If that is the case, the assembly referenced in the method call should be updated to reflect the type's new assembly location.

Category

Windows Forms

Affected APIs

None.

ZipArchiveEntry no longer handles archives with inconsistent entry sizes

Zip archives list both compressed size and uncompressed size in the central directory and local header. The entry data itself also indicates its size. In .NET Core 2.2 and earlier versions, these values were never checked for consistency. Starting with .NET Core 3.0, they now are.

Change description

In .NET Core 2.2 and earlier versions, [ZipArchiveEntry.Open\(\)](#) succeeds even if the local header disagrees with the central header of the zip file. Data is decompressed until the end of the compressed stream is reached, even if its length exceeds the uncompressed file size listed in the central directory/local header.

Starting with .NET Core 3.0, the [ZipArchiveEntry.Open\(\)](#) method checks that local header and central header agree on compressed and uncompressed sizes of an entry. If they do not, the method throws an [InvalidOperationException](#) if the archive's local header and/or data descriptor list sizes that disagree with the central directory of the zip file. When reading an entry, decompressed data is truncated to the uncompressed file size listed in the header.

This change was made to ensure that a [ZipArchiveEntry](#) correctly represents the size of its data and that only that amount of data is read.

Version introduced

3.0

Recommended action

Repackage any zip archive that exhibits these problems.

Category

Core .NET libraries

Affected APIs

- [ZipArchiveEntry.Open\(\)](#)
 - [ZipFileExtensions.ExtractToDirectory](#)
 - [ZipFileExtensions.ExtractToFile](#)
 - [ZipFile.ExtractToDirectory](#)
-

FieldInfo.SetValue throws exception for static, init-only fields

Starting in .NET Core 3.0, an exception is thrown when you attempt to set a value on a static, [InitOnly](#) field by calling [System.Reflection.FieldInfo.SetValue](#).

Change description

In .NET Framework and versions of .NET Core prior to 3.0, you could set the value of a static field that's constant after it is initialized ([readonly in C#](#)) by calling [System.Reflection.FieldInfo.SetValue](#). However, setting such a field in this way resulted in unpredictable behavior based on the target framework and optimization settings.

In .NET Core 3.0 and later versions, when you call [SetValue](#) on a static, [InitOnly](#) field, a [System.FieldAccessException](#) exception is thrown.

 Tip

An `InitOnly` field is one that can only be set at the time it's declared or in the constructor for the containing class. In other words, it's constant after it is initialized.

Version introduced

3.0

Recommended action

Initialize static, `InitOnly` fields in a static constructor. This applies to both dynamic and non-dynamic types.

Alternatively, you can remove the `FieldAttributes.InitOnly` attribute from the field, and then call `FieldInfo.SetValue`.

Category

Core .NET libraries

Affected APIs

- `FieldInfo.SetValue(Object, Object)`
- `FieldInfo.SetValue(Object, Object, BindingFlags, Binder, CultureInfo)`

Passing GroupCollection to extension methods taking `IEnumerable<T>` requires disambiguation

When calling an extension method that takes an `IEnumerable<T>` on a `GroupCollection`, you must disambiguate the type using a cast.

Change description

Starting in .NET Core 3.0, `System.Text.RegularExpressions.GroupCollection` implements `IEnumerable<KeyValuePair<String, Group>>` in addition to the other types it implements, including `IEnumerable<Group>`. This results in ambiguity when calling an extension method that takes an `IEnumerable<T>`. If you call such an extension method on a `GroupCollection` instance, for example, `Enumerable.Count`, you'll see the following compiler error:

CS1061: 'GroupCollection' does not contain a definition for 'Count' and no accessible extension method 'Count' accepting a first argument of type 'GroupCollection' could be found (are you missing a using directive or an assembly reference?)

In previous versions of .NET, there was no ambiguity and no compiler error.

Version introduced

3.0

Reason for change

This was an [unintentional breaking change](#). Because it has been like this for some time, we don't plan to revert it. In addition, such a change would itself be breaking.

Recommended action

For `GroupCollection` instances, disambiguate calls to extension methods that accept an `IEnumerable<T>` with a cast.

C#

```
// Without a cast - causes CS1061.  
match.Groups.Count(_ => true)  
  
// With a disambiguating cast.  
((IEnumerable<Group>)m.Groups).Count(_ => true);
```

Category

Core .NET libraries

Affected APIs

Any extension method that accepts an `IEnumerable<T>` is affected. For example:

- `System.Collections.Immutable.ImmutableArray.TolImmutableArray<TSource>(IEnumerable<TSource>)`
- `System.Collections.Immutable.ImmutableDictionary.TolImmutableDictionary`
- `System.Collections.Immutable.ImmutableHashSet.TolImmutableHashSet`
- `System.Collections.Immutable.ImmutableList.TolImmutableList<TSource>(IEnumerable<TSource>)`

- `System.Collections.Immutable.ImmutableSortedDictionary.ToImmutableSortedDictionary`
 - `System.Collections.Immutable.ImmutableSortedSet.ToImmutableSortedSet`
 - `System.Data.DataTableExtensions.CopyToDataTable`
 - Most of the `System.Linq.Enumerable` methods, for example,
`System.Linq.Enumerable.Count`
 - `System.Linq.ParallelEnumerable.AsParallel`
 - `System.Linq.Queryable.AsQueryable`
-

Cryptography

- BEGIN TRUSTED CERTIFICATE syntax no longer supported on Linux
- EnvelopedCms defaults to AES-256 encryption
- Minimum size for RSAOpenSsl key generation has increased
- .NET Core 3.0 prefers OpenSSL 1.1.x to OpenSSL 1.0.x
- `CryptoStream.Dispose` transforms final block only when writing

"BEGIN TRUSTED CERTIFICATE" syntax no longer supported for root certificates on Linux

Root certificates on Linux and other Unix-like systems (but not macOS) can be presented in two forms: the standard `BEGIN CERTIFICATE` PEM header, and the OpenSSL-specific `BEGIN TRUSTED CERTIFICATE` PEM header. The latter syntax allows for additional configuration that has caused compatibility issues with .NET Core's `System.Security.Cryptography.X509Certificates.X509Chain` class. `BEGIN TRUSTED CERTIFICATE` root certificate contents are no longer loaded by the chain engine starting in .NET Core 3.0.

Change description

Previously, both the `BEGIN CERTIFICATE` and `BEGIN TRUSTED CERTIFICATE` syntaxes were used to populate the root trust list. If the `BEGIN TRUSTED CERTIFICATE` syntax was used and additional options were specified in the file, `X509Chain` may have reported that the chain trust was explicitly disallowed (`X509ChainStatusFlags.ExplicitDistrust`). However, if the certificate was also specified with the `BEGIN CERTIFICATE` syntax in a previously loaded file, the chain trust was allowed.

Starting in .NET Core 3.0, `BEGIN TRUSTED CERTIFICATE` contents are no longer read. If the certificate is not also specified via a standard `BEGIN CERTIFICATE` syntax, the `X509Chain`

reports that the root is not trusted (`X509ChainStatusFlags.UntrustedRoot`).

Version introduced

3.0

Recommended action

Most applications are unaffected by this change, but applications that cannot see both root certificate sources because of permissions problems may experience unexpected `UntrustedRoot` errors after upgrading.

Many Linux distributions (or distros) write root certificates into two locations: a one-certificate-per-file directory, and a one-file concatenation. On some distros, the one-certificate-per-file directory uses the `BEGIN TRUSTED CERTIFICATE` syntax while the file concatenation uses the standard `BEGIN CERTIFICATE` syntax. Ensure that any custom root certificates are added as `BEGIN CERTIFICATE` in at least one of these locations, and that both locations can be read by your application.

The typical directory is `/etc/ssl/certs/` and the typical concatenated file is `/etc/ssl/cert.pem`. Use the command `openssl version -d` to determine the platform-specific root, which may differ from `/etc/ssl/`. For example, on Ubuntu 18.04, the directory is `/usr/lib/ssl/certs/` and the file is `/usr/lib/ssl/cert.pem`. However, `/usr/lib/ssl/certs/` is a symlink to `/etc/ssl/certs/` and `/usr/lib/ssl/cert.pem` does not exist.

Bash

```
$ openssl version -d
OPENSSLDIR: "/usr/lib/ssl"
$ ls -al /usr/lib/ssl
total 12
drwxr-xr-x  3 root root 4096 Dec 12 17:10 .
drwxr-xr-x 73 root root 4096 Feb 20 15:18 ..
lrwxrwxrwx  1 root root   14 Mar 27 2018 certs -> /etc/ssl/certs
drwxr-xr-x  2 root root 4096 Dec 12 17:10 misc
lrwxrwxrwx  1 root root   20 Nov 12 16:58 openssl.cnf ->
/etc/ssl/openssl.cnf
lrwxrwxrwx  1 root root   16 Mar 27 2018 private -> /etc/ssl/private
```

Category

Cryptography

Affected APIs

- [System.Security.Cryptography.X509Certificates.X509Chain](#)
-

EnvelopedCms defaults to AES-256 encryption

The default symmetric encryption algorithm used by `EnvelopedCms` has changed from TripleDES to AES-256.

Change description

In previous versions, when `EnvelopedCms` is used to encrypt data without specifying a symmetric encryption algorithm via a constructor overload, the data is encrypted with the TripleDES/3DES/3DEA/DES3-EDE algorithm.

Starting with .NET Core 3.0 (via version 4.6.0 of the [System.Security.Cryptography.Pkcs](#) NuGet package), the default algorithm has been changed to AES-256 for algorithm modernization and to improve the security of default options. If a message recipient certificate has a (non-EC) Diffie-Hellman public key, the encryption operation may fail with a `CryptographicException` due to limitations in the underlying platform.

In the following sample code, the data is encrypted with TripleDES if running on .NET Core 2.2 or earlier. If running on .NET Core 3.0 or later, it's encrypted with AES-256.

```
C#
```

```
EnvelopedCms cms = new EnvelopedCms(content);
cms.Encrypt(recipient);
return cms.Encode();
```

Version introduced

3.0

Recommended action

If you are negatively impacted by the change, you can restore TripleDES encryption by explicitly specifying the encryption algorithm identifier in an `EnvelopedCms` constructor that includes a parameter of type `AlgorithmIdentifier`, such as:

```
C#
```

```
Oid tripleDesOid = new Oid("1.2.840.113549.3.7", null);
AlgorithmIdentifier tripleDesIdentifier = new
AlgorithmIdentifier(tripleDesOid);
EnvelopedCms cms = new EnvelopedCms(content, tripleDesIdentifier);

cms.Encrypt(recipient);
return cms.Encode();
```

Category

Cryptography

Affected APIs

- [EnvelopedCms\(\)](#)
 - [EnvelopedCms\(ContentInfo\)](#)
 - [EnvelopedCms\(SubjectIdentifierType, ContentInfo\)](#)
-

Minimum size for RSAOpenSsl key generation has increased

The minimum size for generating new RSA keys on Linux has increased from 384-bit to 512-bit.

Change description

Starting with .NET Core 3.0, the minimum legal key size reported by the `LegalKeySizes` property on RSA instances from [RSA.Create](#), [RSAOpenSsl](#), and [RSACryptoServiceProvider](#) on Linux has increased from 384 to 512.

As a result, in .NET Core 2.2 and earlier versions, a method call such as `RSA.Create(384)` succeeds. In .NET Core 3.0 and later versions, the method call `RSA.Create(384)` throws an exception indicating the size is too small.

This change was made because OpenSSL, which performs the cryptographic operations on Linux, raised its minimum between versions 1.0.2 and 1.1.0. .NET Core 3.0 prefers OpenSSL 1.1.x to 1.0.x, and the minimum reported version was raised to reflect this new higher dependency limitation.

Version introduced

Recommended action

If you call any of the affected APIs, ensure that the size of any generated keys is not less than the provider minimum.

Note

384-bit RSA is already considered insecure (as is 512-bit RSA). Modern recommendations, such as [NIST Special Publication 800-57 Part 1 Revision 4](#), suggest 2048-bit as the minimum size for newly generated keys.

Category

Cryptography

Affected APIs

- [AsymmetricAlgorithm.LegalKeySizes](#)
- [RSA.Create](#)
- [RSAOpenSsl](#)
- [RSACryptoServiceProvider](#)

.NET Core 3.0 prefers OpenSSL 1.1.x to OpenSSL 1.0.x

.NET Core for Linux, which works across multiple Linux distributions, can support both OpenSSL 1.0.x and OpenSSL 1.1.x. .NET Core 2.1 and .NET Core 2.2 look for 1.0.x first, then fall back to 1.1.x; .NET Core 3.0 looks for 1.1.x first. This change was made to add support for new cryptographic standards.

This change may impact libraries or applications that do platform interop with the OpenSSL-specific interop types in .NET Core.

Change description

In .NET Core 2.2 and earlier versions, the runtime prefers loading OpenSSL 1.0.x over 1.1.x. This means that the [IntPtr](#) and [SafeHandle](#) types for interop with OpenSSL are used with libcrypto.so.1.0.0 / libcrypto.so.1.0 / libcrypto.so.10 by preference.

Starting with .NET Core 3.0, the runtime prefers loading OpenSSL 1.1.x over OpenSSL 1.0.x, so the [IntPtr](#) and [SafeHandle](#) types for interop with OpenSSL are used with `libcrypto.so.1.1` / `libcrypto.so.11` / `libcrypto.so.1.1.0` / `libcrypto.so.1.1.1` by preference. As a result, libraries and applications that interoperate with OpenSSL directly may have incompatible pointers with the .NET Core-exposed values when upgrading from .NET Core 2.1 or .NET Core 2.2.

Version introduced

3.0

Recommended action

Libraries and applications that do direct operations with OpenSSL need to be careful to ensure they are using the same version of OpenSSL as the .NET Core runtime.

All libraries or applications that use [IntPtr](#) or [SafeHandle](#) values from the .NET Core cryptographic types directly with OpenSSL should compare the version of the library they use with the new [SafeEvpPKeyHandle.OpenSslVersion](#) property to ensure the pointers are compatible.

Category

Cryptography

Affected APIs

- [SafeEvpPKeyHandle](#)
- [RSAOpenSsl\(IntPtr\)](#)
- [RSAOpenSsl\(SafeEvpPKeyHandle\)](#)
- [RSAOpenSsl.DuplicateKeyHandle\(\)](#)
- [DSAOpenSsl\(IntPtr\)](#)
- [DSAOpenSsl\(SafeEvpPKeyHandle\)](#)
- [DSAOpenSsl.DuplicateKeyHandle\(\)](#)
- [ECDsaOpenSsl\(IntPtr\)](#)
- [ECDsaOpenSsl\(SafeEvpPKeyHandle\)](#)
- [ECDsaOpenSsl.DuplicateKeyHandle\(\)](#)
- [ECDiffieHellmanOpenSsl\(IntPtr\)](#)
- [ECDiffieHellmanOpenSsl\(SafeEvpPKeyHandle\)](#)
- [ECDiffieHellmanOpenSsl.DuplicateKeyHandle\(\)](#)
- [X509Certificate.Handle](#)

CryptoStream.Dispose transforms final block only when writing

The [CryptoStream.Dispose](#) method, which is used to finish `CryptoStream` operations, no longer attempts to transform the final block when reading.

Change description

In previous .NET versions, if a user performed an incomplete read when using `CryptoStream` in `Read` mode, the `Dispose` method could throw an exception (for example, when using AES with padding). The exception was thrown because the final block was attempted to be transformed but the data was incomplete.

In .NET Core 3.0 and later versions, `Dispose` no longer tries to transform the final block when reading, which allows for incomplete reads.

Reason for change

This change enables incomplete reads from the crypto stream when a network operation is canceled, without the need to catch an exception.

Version introduced

3.0

Recommended action

Most apps should not be affected by this change.

If your application previously caught an exception in case of an incomplete read, you can delete that `catch` block. If your app used transforming of the final block in hashing scenarios, you might need to ensure that the entire stream is read before it's disposed.

Category

Cryptography

Affected APIs

- [System.Security.Cryptography.CryptoStream.Dispose](#)
-

Entity Framework Core

[Entity Framework Core breaking changes](#)

Globalization

- "C" locale maps to the invariant locale

"C" locale maps to the invariant locale

.NET Core 2.2 and earlier versions depend on the default ICU behavior, which maps the "C" locale to the en_US_POSIX locale. The en_US_POSIX locale has an undesirable collation behavior, because it doesn't support case-insensitive string comparisons. Because some Linux distributions set the "C" locale as the default locale, users were experiencing unexpected behavior.

Change description

Starting with .NET Core 3.0, the "C" locale mapping has changed to use the Invariant locale instead of en_US_POSIX. The "C" locale to Invariant mapping is also applied to Windows for consistency.

Mapping "C" to en_US_POSIX culture caused customer confusion, because en_US_POSIX doesn't support case insensitive sorting/searching string operations. Because the "C" locale is used as a default locale in some of the Linux distros, customers experienced this undesired behavior on these operating systems.

Version introduced

3.0

Recommended action

Nothing specific more than the awareness of this change. This change affects only applications that use the "C" locale mapping.

Category

Affected APIs

All collation and culture APIs are affected by this change.

MSBuild

- [Resource manifest file name change](#)

Resource manifest file name change

Starting in .NET Core 3.0, in the default case, MSBuild generates a different manifest file name for resource files.

Version introduced

3.0

Change description

Prior to .NET Core 3.0, if no `LogicalName`, `ManifestResourceName`, or `DependentUpon` metadata was specified for an `EmbeddedResource` item in the project file, MSBuild generated a manifest file name in the pattern `<RootNamespace>`.

`<ResourceFilePathFromProjectRoot>.resources`. If `RootNamespace` is not defined in the project file, it defaults to the project name. For example, the generated manifest name for a resource file named *Form1.resx* in the root project directory was *MyProject.Form1.resources*.

Starting in .NET Core 3.0, if a resource file is colocated with a source file of the same name (for example, *Form1.resx* and *Form1.cs*), MSBuild uses type information from the source file to generate the manifest file name in the pattern `<Namespace>`.

`<ClassName>.resources`. The namespace and class name are extracted from the first type in the colocated source file. For example, the generated manifest name for a resource file named *Form1.resx* that's colocated with a source file named *Form1.cs* is *MyNamespace.Form1.resources*. The key thing to note is that the first part of the file name is different to prior versions of .NET Core (*MyNamespace* instead of *MyProject*).

 Note

If you have `LogicalName`, `ManifestResourceName`, or `DependentUpon` metadata specified on an `EmbeddedResource` item in the project file, then this change does not affect that resource file.

This breaking change was introduced with the addition of the `EmbeddedResourceUseDependentUponConvention` property to .NET Core projects. By default, resource files aren't explicitly listed in a .NET Core project file, so they have no `DependentUpon` metadata to specify how to name the generated `.resources` file. When `EmbeddedResourceUseDependentUponConvention` is set to `true`, which is the default, MSBuild looks for a colocated source file and extracts a namespace and class name from that file. If you set `EmbeddedResourceUseDependentUponConvention` to `false`, MSBuild generates the manifest name according to the previous behavior, which combines `RootNamespace` and the relative file path.

Recommended action

In most cases, no action is required on the part of the developer, and your app should continue to work. However, if this change breaks your app, you can either:

- Change your code to expect the new manifest name.
- Opt out of the new naming convention by setting `EmbeddedResourceUseDependentUponConvention` to `false` in your project file.

XML

```
<PropertyGroup>
  <EmbeddedResourceUseDependentUponConvention>false</EmbeddedResourceUseDependentUponConvention>
</PropertyGroup>
```

Category

MSBuild

Affected APIs

N/A

Networking

- Default value of `HttpRequestMessage.Version` changed to 1.1

Default value of `HttpRequestMessage.Version` changed to 1.1

The default value of the [System.Net.Http.HttpRequestMessage.Version](#) property has changed from 2.0 to 1.1.

Version introduced

3.0

Change description

In .NET Core 1.0 through 2.0, the default value of the [System.Net.Http.HttpRequestMessage.Version](#) property is 1.1. Starting with .NET Core 2.1, it was changed to 2.1.

Starting with .NET Core 3.0, the default version number returned by the [System.Net.Http.HttpRequestMessage.Version](#) property is once again 1.1.

Recommended action

Update your code if it depends on the [System.Net.Http.HttpRequestMessage.Version](#) property returning a default value of 2.0.

Category

Networking

Affected APIs

- [System.Net.Http.HttpRequestMessage.Version](#)

See also

- [What's new in .NET Core 3.0](#)

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

What's new in .NET Core 2.2

Article • 12/04/2021

.NET Core 2.2 includes enhancements in application deployment, event handling for runtime services, authentication to Azure SQL databases, JIT compiler performance, and code injection prior to the execution of the `Main` method.

New deployment mode

Starting with .NET Core 2.2, you can deploy [framework-dependent executables](#), which are `.exe` files instead of `.dll` files. Functionally similar to framework-dependent deployments, framework-dependent executables (FDE) still rely on the presence of a shared system-wide version of .NET Core to run. Your app contains only your code and any third-party dependencies. Unlike framework-dependent deployments, FDEs are platform-specific.

This new deployment mode has the distinct advantage of building an executable instead of a library, which means you can run your app directly without invoking `dotnet` first.

Core

Handling events in runtime services

You may often want to monitor your application's use of runtime services, such as the GC, JIT, and ThreadPool, to understand how they impact your application. On Windows systems, this is commonly done by monitoring the ETW events of the current process. While this continues to work well, it's not always possible to use ETW if you're running in a low-privilege environment or on Linux or macOS.

Starting with .NET Core 2.2, CoreCLR events can now be consumed using the [System.Diagnostics.Tracing.EventListener](#) class. These events describe the behavior of such runtime services as GC, JIT, ThreadPool, and interop. These are the same events that are exposed as part of the CoreCLR ETW provider. This allows for applications to consume these events or use a transport mechanism to send them to a telemetry aggregation service. You can see how to subscribe to events in the following code sample:

C#

```
internal sealed class SimpleEventListener : EventListener
{
```

```

// Called whenever an EventSource is created.
protected override void OnEventSourceCreated(EventSource eventSource)
{
    // Watch for the .NET runtime EventSource and enable all of its
    events.
    if (eventSource.Name.Equals("Microsoft-Windows-DotNETRuntime"))
    {
        EnableEvents(eventSource, EventLevel.Verbose, (EventKeywords)
(-1));
    }
}

// Called whenever an event is written.
protected override void OnEventWritten(EventWrittenEventArgs eventData)
{
    // Write the contents of the event to the console.
    Console.WriteLine($"ThreadID = {eventData.OSThreadId} ID =
{eventData.EventId} Name = {eventData.EventName}");
    for (int i = 0; i < eventData.Payload.Count; i++)
    {
        string payloadString = eventData.Payload[i]?.ToString() ??
string.Empty;
        Console.WriteLine($" \tName = \'{eventData.PayloadNames[i]}\'\"
Value = \'{payloadString}\''");
    }
    Console.WriteLine("\n");
}
}

```

In addition, .NET Core 2.2 adds the following two properties to the [EventWrittenEventArgs](#) class to provide additional information about ETW events:

- [EventWrittenEventArgs.OSThreadId](#)
- [EventWrittenEventArgs.TimeStamp](#)

Data

[AAD authentication to Azure SQL databases with the SqlConnection.AccessToken property](#)

Starting with .NET Core 2.2, an access token issued by Azure Active Directory can be used to authenticate to an Azure SQL database. To support access tokens, the [AccessToken](#) property has been added to the [SqlConnection](#) class. To take advantage of AAD authentication, download version 4.6 of the System.Data.SqlClient NuGet package. In order to use the feature, you can obtain the access token value using the [Active Directory Authentication Library for .NET](#) contained in the [Microsoft.IdentityModel.Clients.ActiveDirectory](#) NuGet package.

JIT compiler improvements

Tiered compilation remains an opt-in feature

In .NET Core 2.1, the JIT compiler implemented a new compiler technology, *tiered compilation*, as an opt-in feature. The goal of tiered compilation is improved performance. One of the important tasks performed by the JIT compiler is optimizing code execution. For little-used code paths, however, the compiler may spend more time optimizing code than the runtime spends executing unoptimized code. Tiered compilation introduces two stages in JIT compilation:

- A **first tier**, which generates code as quickly as possible.
- A **second tier**, which generates optimized code for those methods that are executed frequently. The second tier of compilation is performed in parallel for enhanced performance.

For information on the performance improvement that can result from tiered compilation, see [Announcing .NET Core 2.2 Preview 2](#).

For information about opting in to tiered compilation, see [Jit compiler improvements](#) in [What's new in .NET Core 2.1](#).

Runtime

Injecting code prior to executing the Main method

Starting with .NET Core 2.2, you can use a startup hook to inject code prior to running an application's Main method. Startup hooks make it possible for a host to customize the behavior of applications after they have been deployed without needing to recompile or change the application.

We expect hosting providers to define custom configuration and policy, including settings that potentially influence the load behavior of the main entry point, such as the [System.Runtime.Loader.AssemblyLoadContext](#) behavior. The hook can be used to set up tracing or telemetry injection, to set up callbacks for handling, or to define other environment-dependent behavior. The hook is separate from the entry point, so that user code doesn't need to be modified.

See [Host startup hook](#) for more information.

See also

- What's new in .NET Core 3.1
- What's new in ASP.NET Core 2.2
- New features in EF Core 2.2

 Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

What's new in .NET Core 2.1

Article • 09/15/2021

.NET Core 2.1 includes enhancements and new features in the following areas:

- [Tooling](#)
- [Roll forward](#)
- [Deployment](#)
- [Windows Compatibility Pack](#)
- [JIT compilation improvements](#)
- [API changes](#)

Tooling

The .NET Core 2.1 SDK (v 2.1.300), the tooling included with .NET Core 2.1, includes the following changes and enhancements:

Build performance improvements

A major focus of .NET Core 2.1 is improving build-time performance, particularly for incremental builds. These performance improvements apply to both command-line builds using `dotnet build` and to builds in Visual Studio. Some individual areas of improvement include:

- For package asset resolution, resolving only assets used by a build rather than all assets.
- Caching of assembly references.
- Use of long-running SDK build servers, which are processes that span across individual `dotnet build` invocations. They eliminate the need to JIT-compile large blocks of code every time `dotnet build` is run. Build server processes can be automatically terminated with the following command:

.NET CLI

```
dotnet buildserver shutdown
```

New CLI commands

A number of tools that were available only on a per project basis using `DotnetCliToolReference` are now available as part of the .NET Core SDK. These tools include:

- `dotnet watch` provides a file system watcher that waits for a file to change before executing a designated set of commands. For example, the following command automatically rebuilds the current project and generates verbose output whenever a file in it changes:

```
.NET CLI
```

```
dotnet watch -- --verbose build
```

Note the `--` option that precedes the `--verbose` option. It delimits the options passed directly to the `dotnet watch` command from the arguments that are passed to the child `dotnet` process. Without it, the `--verbose` option applies to the `dotnet watch` command, not the `dotnet build` command.

For more information, see [Develop ASP.NET Core apps using dotnet watch](#).

- `dotnet dev-certs` generates and manages certificates used during development in ASP.NET Core applications.
- `dotnet user-secrets` manages the secrets in a user secret store in ASP.NET Core applications.
- `dotnet sql-cache` creates a table and indexes in a Microsoft SQL Server database to be used for distributed caching.
- `dotnet ef` is a tool for managing databases, `DbContext` objects, and migrations in Entity Framework Core applications. For more information, see [EF Core .NET Command-line Tools](#).

Global Tools

.NET Core 2.1 supports *Global Tools* -- that is, custom tools that are available globally from the command line. The extensibility model in previous versions of .NET Core made custom tools available on a per project basis only by using `DotnetCliToolReference`.

To install a Global Tool, you use the `dotnet tool install` command. For example:

```
.NET CLI
```

```
dotnet tool install -g dotnetsay
```

Once installed, the tool can be run from the command line by specifying the tool name. For more information, see [.NET Core Global Tools overview](#).

Tool management with the `dotnet tool` command

In .NET Core 2.1 SDK, all tools operations use the `dotnet tool` command. The following options are available:

- [dotnet tool install](#) to install a tool.
- [dotnet tool update](#) to uninstall and reinstall a tool, which effectively updates it.
- [dotnet tool list](#) to list currently installed tools.
- [dotnet tool uninstall](#) to uninstall currently installed tools.

Roll forward

All .NET Core applications starting with .NET Core 2.0 automatically roll forward to the latest *minor version* installed on a system.

Starting with .NET Core 2.0, if the version of .NET Core that an application was built with is not present at run time, the application automatically runs against the latest installed *minor version* of .NET Core. In other words, if an application is built with .NET Core 2.0, and .NET Core 2.0 is not present on the host system but .NET Core 2.1 is, the application runs with .NET Core 2.1.

Important

This roll-forward behavior doesn't apply to preview releases. By default, it also doesn't apply to major releases, but this can be changed with the settings below.

You can modify this behavior by changing the setting for the roll-forward on no candidate shared framework. The available settings are:

- `0` - disable minor version roll-forward behavior. With this setting, an application built for .NET Core 2.0.0 will roll forward to .NET Core 2.0.1, but not to .NET Core 2.2.0 or .NET Core 3.0.0.

- 1 - enable minor version roll-forward behavior. This is the default value for the setting. With this setting, an application built for .NET Core 2.0.0 will roll forward to either .NET Core 2.0.1 or .NET Core 2.2.0, depending on which one is installed, but it will not roll forward to .NET Core 3.0.0.
- 2 - enable minor and major version roll-forward behavior. If set, even different major versions are considered, so an application built for .NET Core 2.0.0 will roll forward to .NET Core 3.0.0.

You can modify this setting in any of three ways:

- Set the `DOTNET_ROLL_FORWARD_ON_NO_CANDIDATE_FX` environment variable to the desired value.
- Add the following line with the desired value to the `.runtimeconfig.json` file:

JSON

```
"rollForwardOnNoCandidateFx" : 0
```

- When using the [.NET Core CLI](#), add the following option with the desired value to a .NET Core command such as `run`:

.NET CLI

```
dotnet run --rollForwardOnNoCandidateFx=0
```

Patch version roll forward is independent of this setting and is done after any potential minor or major version roll forward is applied.

Deployment

Self-contained application servicing

`dotnet publish` now publishes self-contained applications with a serviced runtime version. When you publish a self-contained application with the .NET Core 2.1 SDK (v 2.1.300), your application includes the latest serviced runtime version known by that SDK. When you upgrade to the latest SDK, you'll publish with the latest .NET Core runtime version. This applies for .NET Core 1.0 runtimes and later.

Self-contained publishing relies on runtime versions on NuGet.org. You do not need to have the serviced runtime on your machine.

Using the .NET Core 2.0 SDK, self-contained applications are published with the .NET Core 2.0.0 runtime unless a different version is specified via the `RuntimeFrameworkVersion` property. With this new behavior, you'll no longer need to set this property to select a higher runtime version for a self-contained application. The easiest approach going forward is to always publish with .NET Core 2.1 SDK (v 2.1.300).

For more information, see [Self-contained deployment runtime roll forward](#).

Windows Compatibility Pack

When you port existing code from the .NET Framework to .NET Core, you can use the [Windows Compatibility Pack](#). It provides access to 20,000 more APIs than are available in .NET Core. These APIs include types in the `System.Drawing` namespace, the `EventLog` class, WMI, Performance Counters, Windows Services, and the Windows registry types and members.

JIT compiler improvements

.NET Core incorporates a new JIT compiler technology called *tiered compilation* (also known as *adaptive optimization*) that can significantly improve performance. Tiered compilation is an opt-in setting.

One of the important tasks performed by the JIT compiler is optimizing code execution. For little-used code paths, however, the compiler may spend more time optimizing code than the runtime spends running unoptimized code. Tiered compilation introduces two stages in JIT compilation:

- A **first tier**, which generates code as quickly as possible.
- A **second tier**, which generates optimized code for those methods that are executed frequently. The second tier of compilation is performed in parallel for enhanced performance.

You can opt into tiered compilation in either of two ways.

- To use tiered compilation in all projects that use the .NET Core 2.1 SDK, set the following environment variable:

Console

```
COMPlus_TieredCompilation="1"
```

- To use tiered compilation on a per-project basis, add the `<TieredCompilation>` property to the `<PropertyGroup>` section of the MSBuild project file, as the following example shows:

XML

```
<PropertyGroup>
  <!-- other property definitions -->

  <TieredCompilation>true</TieredCompilation>
</PropertyGroup>
```

API changes

Span<T> and Memory<T>

.NET Core 2.1 includes some new types that make working with arrays and other types of memory much more efficient. The new types include:

- `System.Span<T>` and `System.ReadOnlySpan<T>`.
- `System.Memory<T>` and `System.ReadOnlyMemory<T>`.

Without these types, when passing such items as a portion of an array or a section of a memory buffer, you have to make a copy of some portion of the data before passing it to a method. These types provide a virtual view of that data that eliminates the need for the additional memory allocation and copy operations.

The following example uses a `Span<T>` and `Memory<T>` instance to provide a virtual view of 10 elements of an array.

C#

```
using System;

class Program
{
    static void Main()
    {
        int[] numbers = new int[100];
        for (int i = 0; i < 100; i++)
        {
            numbers[i] = i * 2;
        }

        var part = new Span<int>(numbers, start: 10, length: 10);
```

```
        foreach (var value in part)
            Console.WriteLine($"{value} ");
    }
}
// The example displays the following output:
//      20 22 24 26 28 30 32 34 36 38
```

Brotli compression

.NET Core 2.1 adds support for Brotli compression and decompression. Brotli is a general-purpose lossless compression algorithm that is defined in [RFC 7932](#) and is supported by most web browsers and major web servers. You can use the stream-based [System.IO.Compression.BrotliStream](#) class or the high-performance span-based [System.IO.Compression.BrotliEncoder](#) and [System.IO.Compression.BrotliDecoder](#) classes. The following example illustrates compression with the [BrotliStream](#) class:

C#

```
public static Stream DecompressWithBrotli(Stream toDecompress)
{
    MemoryStream decompressedStream = new MemoryStream();
    using (BrotliStream decompressionStream = new BrotliStream(toDecompress,
CompressionMode.Decompress))
    {
        decompressionStream.CopyTo(decompressedStream);
    }
    decompressedStream.Position = 0;
    return decompressedStream;
}
```

The [BrotliStream](#) behavior is the same as [DeflateStream](#) and [GZipStream](#), which makes it easy to convert code that calls these APIs to [BrotliStream](#).

New cryptography APIs and cryptography improvements

.NET Core 2.1 includes numerous enhancements to the cryptography APIs:

- [System.Security.Cryptography.Pkcs.SignedCms](#) is available in the [System.Security.Cryptography.Pkcs](#) package. The implementation is the same as the [SignedCms](#) class in the .NET Framework.
- New overloads of the [X509Certificate.GetCertHash](#) and [X509Certificate.GetCertHashString](#) methods accept a hash algorithm identifier to enable callers to get certificate thumbprint values using algorithms other than SHA-1.

- New `Span<T>`-based cryptography APIs are available for hashing, HMAC, cryptographic random number generation, asymmetric signature generation, asymmetric signature processing, and RSA encryption.
- The performance of `System.Security.Cryptography.Rfc2898DeriveBytes` has improved by about 15% by using a `Span<T>`-based implementation.
- The new `System.Security.Cryptography.CryptographicOperations` class includes two new methods:
 - `FixedTimeEquals` takes a fixed amount of time to return for any two inputs of the same length, which makes it suitable for use in cryptographic verification to avoid contributing to timing side-channel information.
 - `ZeroMemory` is a memory-clearing routine that cannot be optimized.
- The static `RandomNumberGenerator.Fill` method fills a `Span<T>` with random values.
- The `System.Security.Cryptography.Pkcs.EnvelopedCms` is now supported on Linux and macOS.
- Elliptic-Curve Diffie-Hellman (ECDH) is now available in the `System.Security.Cryptography.ECDiffieHellman` class family. The surface area is the same as in the .NET Framework.
- The instance returned by `RSA.Create` can encrypt or decrypt with OAEP using a SHA-2 digest, as well as generate or validate signatures using RSA-PSS.

Sockets improvements

.NET Core includes a new type, `System.Net.Http.SocketsHttpHandler`, and a rewritten `System.Net.Http.HttpMessageHandler`, that form the basis of higher-level networking APIs. `System.Net.Http.SocketsHttpHandler`, for example, is the basis of the `HttpClient` implementation. In previous versions of .NET Core, higher-level APIs were based on native networking implementations.

The sockets implementation introduced in .NET Core 2.1 has a number of advantages:

- A significant performance improvement when compared with the previous implementation.
- Elimination of platform dependencies, which simplifies deployment and servicing.
- Consistent behavior across all .NET Core platforms.

[SocketsHttpHandler](#) is the default implementation in .NET Core 2.1. However, you can configure your application to use the older [HttpClientHandler](#) class by calling the [AppContext.SetSwitch](#) method:

C#

```
AppContext.SetSwitch("System.Net.Http.UseSocketsHttpHandler", false);
```

You can also use an environment variable to opt out of using sockets implementations based on [SocketsHttpHandler](#). To do this, set the

`DOTNET_SYSTEM_NET_HTTP_USESOCKETSHTTPHANDLER` to either `false` or 0.

On Windows, you can also choose to use [System.Net.Http.WinHttpHandler](#), which relies on a native implementation, or the [SocketsHttpHandler](#) class by passing an instance of the class to the [HttpClient](#) constructor.

On Linux and macOS, you can only configure [HttpClient](#) on a per-process basis. On Linux, you need to deploy [libcurl](#) if you want to use the old [HttpClient](#) implementation. (It is installed with .NET Core 2.0.)

Breaking changes

For information about breaking changes, see [Breaking changes for migration from version 2.0 to 2.1](#).

See also

- [What's new in .NET Core 3.1](#)
- [New features in EF Core 2.1](#)
- [What's new in ASP.NET Core 2.1](#)

 Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

 .NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

Breaking changes in .NET Core 2.1

Article • 03/18/2023

If you're migrating to version 2.1 of .NET Core, the breaking changes listed in this article may affect your app.

Core .NET libraries

- Path APIs don't throw an exception for invalid characters
- Private fields added to built-in struct types
- OpenSSL versions on macOS

Path APIs don't throw an exception for invalid characters

APIs that involve file paths no longer validate path characters or throw an [ArgumentException](#) if an invalid character is found.

Change description

In .NET Framework and .NET Core 1.0 - 2.0, the methods listed in the [Affected APIs](#) section throw an [ArgumentException](#) if the path argument contains an invalid path character. Starting in .NET Core 2.1, these methods no longer check for [invalid path characters](#) or throw an exception if an invalid character is found.

Reason for change

Aggressive validation of path characters blocks some cross-platform scenarios. This change was introduced so that .NET does not try to replicate or predict the outcome of operating system API calls. For more information, see the [System.IO in .NET Core 2.1 sneak peek](#) blog post.

Version introduced

.NET Core 2.1

Recommended action

If your code relied on these APIs to check for invalid characters, you can add a call to [Path.GetInvalidPathChars](#).

Affected APIs

- [System.IO.Directory.CreateDirectory](#)
- [System.IO.Directory.Delete](#)
- [System.IO.Directory.EnumerateDirectories](#)
- [System.IO.Directory.EnumerateFiles](#)
- [System.IO.Directory.EnumerateFileSystemEntries](#)
- [System.IO.Directory.GetCreationTime\(String\)](#)
- [System.IO.Directory.GetCreationTimeUtc\(String\)](#)
- [System.IO.Directory.GetDirectories](#)
- [System.IO.Directory.GetDirectoryRoot\(String\)](#)
- [System.IO.Directory.GetFiles](#)
- [System.IO.Directory.GetFileSystemEntries](#)
- [System.IO.Directory.GetLastAccessTime\(String\)](#)
- [System.IO.Directory.GetLastAccessTimeUtc\(String\)](#)
- [System.IO.Directory.GetLastWriteTime\(String\)](#)
- [System.IO.Directory.GetLastWriteTimeUtc\(String\)](#)
- [System.IO.Directory.GetParent\(String\)](#)
- [System.IO.Directory.Move\(String, String\)](#)
- [System.IO.Directory.SetCreationTime\(String, DateTime\)](#)
- [System.IO.Directory.SetCreationTimeUtc\(String, DateTime\)](#)
- [System.IO.Directory.SetCurrentDirectory\(String\)](#)
- [System.IO.Directory.SetLastAccessTime\(String, DateTime\)](#)
- [System.IO.Directory.SetLastAccessTimeUtc\(String, DateTime\)](#)
- [System.IO.Directory.SetLastWriteTime\(String, DateTime\)](#)
- [System.IO.Directory.SetLastWriteTimeUtc\(String, DateTime\)](#)
- [System.IO.DirectoryInfo ctor](#)
- [System.IO.Directory.GetDirectories](#)
- [System.IO.Directory.GetFiles](#)
- [System.IO.DirectoryInfo.GetFileSystemInfos](#)
- [System.IO.File.AppendAllText](#)
- [System.IO.File.AppendAllTextAsync](#)
- [System.IO.File.Copy](#)
- [System.IO.File.Create](#)
- [System.IO.File.CreateText](#)
- [System.IO.File.Decrypt](#)
- [System.IO.File.Delete](#)
- [System.IO.File.Encrypt](#)
- [System.IO.File.GetAttributes\(String\)](#)
- [System.IO.File.GetCreationTime\(String\)](#)
- [System.IO.File.GetCreationTimeUtc\(String\)](#)

- [System.IO.File.GetLastAccessTime\(String\)](#)
- [System.IO.File.GetLastAccessTimeUtc\(String\)](#)
- [System.IO.File.GetLastWriteTime\(String\)](#)
- [System.IO.File.GetLastWriteTimeUtc\(String\)](#)
- [System.IO.File.Move](#)
- [System.IO.File.Open](#)
- [System.IO.File.OpenRead\(String\)](#)
- [System.IO.File.OpenText\(String\)](#)
- [System.IO.File.OpenWrite\(String\)](#)
- [System.IO.File.ReadAllBytes\(String\)](#)
- [System.IO.File.ReadAllBytesAsync\(String, CancellationToken\)](#)
- [System.IO.File.ReadAllLines\(String\)](#)
- [System.IO.File.ReadAllLinesAsync\(String, CancellationToken\)](#)
- [System.IO.File.ReadAllText\(String\)](#)
- [System.IO.File.ReadAllTextAsync\(String, CancellationToken\)](#)
- [System.IO.File.SetAttributes\(String, FileAttributes\)](#)
- [System.IO.File.SetCreationTime\(String, DateTime\)](#)
- [System.IO.File.SetCreationTimeUtc\(String, DateTime\)](#)
- [System.IO.File.SetLastAccessTime\(String, DateTime\)](#)
- [System.IO.File.SetLastAccessTimeUtc\(String, DateTime\)](#)
- [System.IO.File.SetLastWriteTime\(String, DateTime\)](#)
- [System.IO.File.SetLastWriteTimeUtc\(String, DateTime\)](#)
- [System.IO.File.WriteAllBytes\(String, Byte\[\]\)](#)
- [System.IO.File.WriteAllBytesAsync\(String, Byte\[\], CancellationToken\)](#)
- [System.IO.File.WriteAllLines](#)
- [System.IO.File.WriteAllLinesAsync](#)
- [System.IO.File.WriteAllText](#)
- [System.IO.FileInfo ctor](#)
- [System.IO.FileInfo.CopyTo](#)
- [System.IO.FileInfo.MoveTo](#)
- [System.IO.FileStream ctor](#)
- [System.IO.Path.GetFullPath\(String\)](#)
- [System.IO.Path.IsPathRooted\(String\)](#)
- [System.IO.Path.GetPathRoot\(String\)](#)
- [System.IO.Path.ChangeExtension\(String, String\)](#)
- [System.IO.Path.GetDirectoryName\(String\)](#)
- [System.IO.Path.GetExtension\(String\)](#)
- [System.IO.Path.HasExtension\(String\)](#)
- [System.IO.Path.Combine](#)

See also

- [System.IO in .NET Core 2.1 sneak peek](#)

Private fields added to built-in struct types

Private fields were added to [certain struct types](#) in [reference assemblies](#). As a result, in C#, those struct types must always be instantiated by using the [new operator](#) or [default literal](#).

Change description

In .NET Core 2.0 and previous versions, some provided struct types, for example, [ConsoleKeyInfo](#), could be instantiated without using the [new](#) operator or [default literal](#) in C#. This was because the [reference assemblies](#) used by the C# compiler didn't contain the private fields for the structs. All private fields for .NET struct types are added to the reference assemblies starting in .NET Core 2.1.

For example, the following C# code compiles in .NET Core 2.0, but not in .NET Core 2.1:

```
C#  
  
ConsoleKeyInfo key;      // Struct type  
  
if (key.ToString() == "y")  
{  
    Console.WriteLine("Yes!");  
}
```

In .NET Core 2.1, the previous code results in the following compiler error: **CS0165 - Use of unassigned local variable 'key'**

Version introduced

2.1

Recommended action

Instantiate struct types by using the [new](#) operator or [default literal](#).

For example:

```
C#
```

```
ConsoleKeyInfo key = new ConsoleKeyInfo(); // Struct type.  
  
if (key.ToString() == "y")  
    Console.WriteLine("Yes!");
```

```
C#
```

```
ConsoleKeyInfo key = default; // Struct type.  
  
if (key.ToString() == "y")  
    Console.WriteLine("Yes!");
```

Category

Core .NET libraries

Affected APIs

- [System.ArraySegment<T>.Enumerator](#)
- [System.ArraySegment<T>](#)
- [System.Boolean](#)
- [System.Buffers.MemoryHandle](#)
- [System.Buffers.StandardFormat](#)
- [System.Byte](#)
- [System.Char](#)
- [System.Collections.DictionaryEntry](#)
- [System.Collections.Generic.Dictionary<TKey,TValue>.Enumerator](#)
- [System.Collections.Generic.Dictionary<TKey,TValue>.KeyCollection.Enumerator](#)
- [System.Collections.Generic.Dictionary<TKey,TValue>.ValueCollection.Enumerator](#)
- [System.Collections.Generic.HashSet<T>.Enumerator](#)
- [System.Collections.Generic.KeyValuePair<TKey,TValue>](#)
- [System.Collections.Generic.LinkedList<T>.Enumerator](#)
- [System.Collections.Generic.List<T>.Enumerator](#)
- [System.Collections.Generic.Queue<T>.Enumerator](#)
- [System.Collections.Generic.SortedDictionary<TKey,TValue>.Enumerator](#)
- [System.Collections.Generic.SortedDictionary<TKey,TValue>.KeyCollection.Enumerator](#)
- [System.Collections.Generic.SortedDictionary<TKey,TValue>.ValueCollection.Enumerator](#)
- [System.Collections.Generic.SortedSet<T>.Enumerator](#)

- [System.Collections.Generic.Stack<T>.Enumerator](#)
- [System.Collections.Immutable.ImmutableArray<T>.Enumerator](#)
- [System.Collections.Immutable.ImmutableArray<T>](#)
- [System.Collections.Immutable.ImmutableDictionary<TKey,TValue>.Enumerator](#)
- [System.Collections.Immutable.ImmutableHashSet<T>.Enumerator](#)
- [System.Collections.Immutable.ImmutableList<T>.Enumerator](#)
- [System.Collections.Immutable.ImmutableQueue<T>.Enumerator](#)
- [System.Collections.Immutable.ImmutableSortedDictionary<TKey,TValue>.Enumerator](#)
or
- [System.Collections.Immutable.ImmutableSortedSet<T>.Enumerator](#)
- [System.Collections.Immutable.ImmutableStack<T>.Enumerator](#)
- [System.Collections.Specialized.BitVector32.Section](#)
- [System.Collections.Specialized.BitVector32](#)
- [LazyMemberInfo](#)
- [System.ComponentModel.Design.Serialization.MemberRelationship](#)
- [System.ConsoleKeyInfo](#)
- [System.Data.SqlTypes.SqlBinary](#)
- [System.Data.SqlTypes.SqlBoolean](#)
- [System.Data.SqlTypes.SqlByte](#)
- [System.Data.SqlTypes.SqlDateTime](#)
- [System.Data.SqlTypes.SqlDecimal](#)
- [System.Data.SqlTypes.SqlDouble](#)
- [System.Data.SqlTypes.SqlGuid](#)
- [System.Data.SqlTypes.SqlInt16](#)
- [System.Data.SqlTypes.SqlInt32](#)
- [System.Data.SqlTypes.SqlInt64](#)
- [System.Data.SqlTypes.SqlMoney](#)
- [System.Data.SqlTypes.SqlSingle](#)
- [System.Data.SqlTypes.SqlString](#)
- [System.DateTime](#)
- [System DateTimeOffset](#)
- [System.Decimal](#)
- [System.Diagnostics.CounterSample](#)
- [System.Diagnostics.SymbolStore.SymbolToken](#)
- [System.Diagnostics.Tracing.EventSourceEventData](#)
- [System.Diagnostics.Tracing.EventSourceOptions](#)
- [System.Double](#)
- [System.Drawing.CharacterRange](#)
- [System.Drawing.Point](#)
- [System.Drawing.PointF](#)

- [System.Drawing.Rectangle](#)
- [System.Drawing.RectangleF](#)
- [System.Drawing.Size](#)
- [System.Drawing.SizeF](#)
- [System.Guid](#)
- [System.HashCode](#)
- [System.Int16](#)
- [System.Int32](#)
- [System.Int64](#)
- [System.IntPtr](#)
- [System.IO.Pipelines.FlushResult](#)
- [System.IO.Pipelines.ReadResult](#)
- [System.IO.WaitForChangedResult](#)
- [System.Memory<T>](#)
- [System.ModuleHandle](#)
- [System.Net.Security.SslApplicationProtocol](#)
- [System.Net.Sockets.IPPacketInformation](#)
- [System.Net.Sockets.SocketInformation](#)
- [System.Net.Sockets.UdpReceiveResult](#)
- [System.Net.WebSockets.ValueWebSocketReceiveResult](#)
- [System.Nullable<T>](#)
- [System.Numerics.BigInteger](#)
- [System.Numerics.Complex](#)
- [System.Numerics.Vector<T>](#)
- [System.ReadOnlyMemory<T>](#)
- [System.ReadOnlySpan<T>.Enumerator](#)
- [System.ReadOnlySpan<T>](#)
- [System.Reflection.CustomAttributeNamedArgument](#)
- [System.Reflection.CustomAttributeTypedArgument](#)
- [System.Reflection.Emit.Label](#)
- [System.Reflection.Emit.OpCode](#)
- [System.Reflection.Metadata.ArrayShape](#)
- [System.Reflection.Metadata.AssemblyDefinition](#)
- [System.Reflection.Metadata.AssemblyDefinitionHandle](#)
- [System.Reflection.Metadata.AssemblyFile](#)
- [System.Reflection.Metadata.AssemblyFileHandle](#)
- [System.Reflection.Metadata.AssemblyFileHandleCollection.Enumerator](#)
- [System.Reflection.Metadata.AssemblyFileHandleCollection](#)
- [System.Reflection.Metadata.AssemblyReference](#)
- [System.Reflection.Metadata.AssemblyReferenceHandle](#)

- [System.Reflection.Metadata.AssemblyReferenceHandleCollection.Enumerator](#)
- [System.Reflection.Metadata.AssemblyReferenceHandleCollection](#)
- [System.Reflection.Metadata.Blob](#)
- [System.Reflection.Metadata.BlobBuilder.Blobs](#)
- [System.Reflection.Metadata.BlobContentId](#)
- [System.Reflection.Metadata.BlobHandle](#)
- [System.Reflection.Metadata.BlobReader](#)
- [System.Reflection.Metadata.BlobWriter](#)
- [System.Reflection.Metadata.Constant](#)
- [System.Reflection.Metadata.ConstantHandle](#)
- [System.Reflection.Metadata.CustomAttribute](#)
- [System.Reflection.Metadata.CustomAttributeHandle](#)
- [System.Reflection.Metadata.CustomAttributeHandleCollection.Enumerator](#)
- [System.Reflection.Metadata.CustomAttributeHandleCollection](#)
- [System.Reflection.Metadata.CustomAttributeNamedArgument<TType>](#)
- [System.Reflection.Metadata.CustomAttributeTypedArgument<TType>](#)
- [System.Reflection.Metadata.CustomAttributeValue<TType>](#)
- [System.Reflection.Metadata.CustomDebugInformation](#)
- [System.Reflection.Metadata.CustomDebugInformationHandle](#)
- [System.Reflection.Metadata.CustomDebugInformationHandleCollection.Enumerator](#)
- [System.Reflection.Metadata.CustomDebugInformationHandleCollection](#)
- [System.Reflection.Metadata.DeclarativeSecurityAttribute](#)
- [System.Reflection.Metadata.DeclarativeSecurityAttributeHandle](#)
- [System.Reflection.Metadata.DeclarativeSecurityAttributeHandleCollection.Enumerator](#)
- [System.Reflection.Metadata.DeclarativeSecurityAttributeHandleCollection](#)
- [System.Reflection.Metadata.Document](#)
- [System.Reflection.Metadata.DocumentHandle](#)
- [System.Reflection.Metadata.DocumentHandleCollection.Enumerator](#)
- [System.Reflection.Metadata.DocumentHandleCollection](#)
- [System.Reflection.Metadata.DocumentNameBlobHandle](#)
- [System.Reflection.Metadata.Ecma335.ArrayShapeEncoder](#)
- [System.Reflection.Metadata.Ecma335.BlobEncoder](#)
- [System.Reflection.Metadata.Ecma335.CustomAttributeArrayTypeEncoder](#)
- [System.Reflection.Metadata.Ecma335.CustomAttributeElementTypeEncoder](#)
- [System.Reflection.Metadata.Ecma335.CustomAttributeNamedArgumentsEncoder](#)
- [System.Reflection.Metadata.Ecma335.CustomModifiersEncoder](#)
- [System.Reflection.Metadata.Ecma335.EditAndContinueLogEntry](#)
- [System.Reflection.Metadata.Ecma335.ExceptionRegionEncoder](#)

- [System.Reflection.Metadata.Ecma335.FixedArgumentsEncoder](#)
- [System.Reflection.Metadata.Ecma335.GenericTypeArgumentsEncoder](#)
- [System.Reflection.Metadata.Ecma335.InstructionEncoder](#)
- [System.Reflection.Metadata.Ecma335.LabelHandle](#)
- [System.Reflection.Metadata.Ecma335.LiteralEncoder](#)
- [System.Reflection.Metadata.Ecma335.LiteralsEncoder](#)
- [System.Reflection.Metadata.Ecma335.LocalVariablesEncoder](#)
- [System.Reflection.Metadata.Ecma335.LocalVariableTypeEncoder](#)
- [System.Reflection.Metadata.Ecma335.MethodBodyStreamEncoder.MethodBody](#)
- [System.Reflection.Metadata.Ecma335.MethodBodyStreamEncoder](#)
- [System.Reflection.Metadata.Ecma335.MethodSignatureEncoder](#)
- [System.Reflection.Metadata.Ecma335.NamedArgumentsEncoder](#)
- [System.Reflection.Metadata.Ecma335.NamedArgumentTypeEncoder](#)
- [System.Reflection.Metadata.Ecma335.NameEncoder](#)
- [System.Reflection.Metadata.Ecma335.ParametersEncoder](#)
- [System.Reflection.Metadata.Ecma335.ParameterTypeEncoder](#)
- [System.Reflection.Metadata.Ecma335.PermissionSetEncoder](#)
- [System.Reflection.Metadata.Ecma335.ReturnTypeEncoder](#)
- [System.Reflection.Metadata.Ecma335.ScalarEncoder](#)
- [System.Reflection.Metadata.Ecma335.SignatureDecoder<TTyp, TGenericContext>](#)
- [System.Reflection.Metadata.Ecma335.SignatureTypeEncoder](#)
- [System.Reflection.Metadata.Ecma335.VectorEncoder](#)
- [System.Reflection.Metadata.EntityHandle](#)
- [System.Reflection.Metadata.EventAccessors](#)
- [System.Reflection.Metadata.EventDefinition](#)
- [System.Reflection.Metadata.EventDefinitionHandle](#)
- [System.Reflection.Metadata.EventDefinitionHandleCollection.Enumerator](#)
- [System.Reflection.Metadata.EventDefinitionHandleCollection](#)
- [System.Reflection.Metadata.ExceptionRegion](#)
- [System.Reflection.Metadata.ExportedType](#)
- [System.Reflection.Metadata.ExportedTypeHandle](#)
- [System.Reflection.Metadata.ExportedTypeHandleCollection.Enumerator](#)
- [System.Reflection.Metadata.ExportedTypeHandleCollection](#)
- [System.Reflection.Metadata.FieldDefinition](#)
- [System.Reflection.Metadata.FieldDefinitionHandle](#)
- [System.Reflection.Metadata.FieldDefinitionHandleCollection.Enumerator](#)
- [System.Reflection.Metadata.FieldDefinitionHandleCollection](#)
- [System.Reflection.Metadata.GenericParameter](#)
- [System.Reflection.Metadata.GenericParameterConstraint](#)
- [System.Reflection.Metadata.GenericParameterConstraintHandle](#)

- [System.Reflection.Metadata.GenericParameterConstraintHandleCollection.Enumerator](#)
- [System.Reflection.Metadata.GenericParameterConstraintHandleCollection](#)
- [System.Reflection.Metadata.GenericParameterHandle](#)
- [System.Reflection.Metadata.GenericParameterHandleCollection.Enumerator](#)
- [System.Reflection.Metadata.GenericParameterHandleCollection](#)
- [System.Reflection.Metadata.GuidHandle](#)
- [System.Reflection.Metadata.Handle](#)
- [System.Reflection.Metadata.ImportDefinition](#)
- [System.Reflection.Metadata.ImportDefinitionCollection.Enumerator](#)
- [System.Reflection.Metadata.ImportDefinitionCollection](#)
- [System.Reflection.Metadata.ImportScope](#)
- [System.Reflection.Metadata.ImportScopeCollection.Enumerator](#)
- [System.Reflection.Metadata.ImportScopeCollection](#)
- [System.Reflection.Metadata.ImportScopeHandle](#)
- [System.Reflection.Metadata.InterfaceImplementation](#)
- [System.Reflection.Metadata.InterfaceImplementationHandle](#)
- [System.Reflection.Metadata.InterfaceImplementationHandleCollection.Enumerator](#)
- [System.Reflection.Metadata.InterfaceImplementationHandleCollection](#)
- [System.Reflection.Metadata.LocalConstant](#)
- [System.Reflection.Metadata.LocalConstantHandle](#)
- [System.Reflection.Metadata.LocalConstantHandleCollection.Enumerator](#)
- [System.Reflection.Metadata.LocalConstantHandleCollection](#)
- [System.Reflection.Metadata.LocalScope](#)
- [System.Reflection.Metadata.LocalScopeHandle](#)
- [System.Reflection.Metadata.LocalScopeHandleCollection.ChildrenEnumerator](#)
- [System.Reflection.Metadata.LocalScopeHandleCollection.Enumerator](#)
- [System.Reflection.Metadata.LocalScopeHandleCollection](#)
- [System.Reflection.Metadata.LocalVariable](#)
- [System.Reflection.Metadata.LocalVariableHandle](#)
- [System.Reflection.Metadata.LocalVariableHandleCollection.Enumerator](#)
- [System.Reflection.Metadata.LocalVariableHandleCollection](#)
- [System.Reflection.Metadata.ManifestResource](#)
- [System.Reflection.Metadata.ManifestResourceHandle](#)
- [System.Reflection.Metadata.ManifestResourceHandleCollection.Enumerator](#)
- [System.Reflection.Metadata.ManifestResourceHandleCollection](#)
- [System.Reflection.Metadata.MemberReference](#)
- [System.Reflection.Metadata.MemberReferenceHandle](#)
- [System.Reflection.Metadata.MemberReferenceHandleCollection.Enumerator](#)
- [System.Reflection.Metadata.MemberReferenceHandleCollection](#)

- [System.Reflection.Metadata.MetadataStringComparer](#)
- [System.Reflection.Metadata.MethodDebugInformation](#)
- [System.Reflection.Metadata.MethodDebugInformationHandle](#)
- [System.Reflection.Metadata.MethodDebugInformationHandleCollection.Enumerator](#)
- [System.Reflection.Metadata.MethodDebugInformationHandleCollection](#)
- [System.Reflection.Metadata.MethodDefinition](#)
- [System.Reflection.Metadata.MethodDefinitionHandle](#)
- [System.Reflection.Metadata.MethodDefinitionHandleCollection.Enumerator](#)
- [System.Reflection.Metadata.MethodDefinitionHandleCollection](#)
- [System.Reflection.Metadata.MethodImplementation](#)
- [System.Reflection.Metadata.MethodImplementationHandle](#)
- [System.Reflection.Metadata.MethodImplementationHandleCollection.Enumerator](#)
- [System.Reflection.Metadata.MethodImplementationHandleCollection](#)
- [System.Reflection.Metadata.MethodImport](#)
- [System.Reflection.Metadata.MethodSignature<TType>](#)
- [System.Reflection.Metadata.MethodSpecification](#)
- [System.Reflection.Metadata.MethodSpecificationHandle](#)
- [System.Reflection.Metadata.ModuleDefinition](#)
- [System.Reflection.Metadata.ModuleDefinitionHandle](#)
- [System.Reflection.Metadata.ModuleReference](#)
- [System.Reflection.Metadata.ModuleReferenceHandle](#)
- [System.Reflection.Metadata.NamespaceDefinition](#)
- [System.Reflection.Metadata.NamespaceDefinitionHandle](#)
- [System.Reflection.Metadata.Parameter](#)
- [System.Reflection.Metadata.ParameterHandle](#)
- [System.Reflection.Metadata.ParameterHandleCollection.Enumerator](#)
- [System.Reflection.Metadata.ParameterHandleCollection](#)
- [System.Reflection.Metadata.PropertyAccessors](#)
- [System.Reflection.Metadata.PropertyDefinition](#)
- [System.Reflection.Metadata.PropertyDefinitionHandle](#)
- [System.Reflection.Metadata.PropertyDefinitionHandleCollection.Enumerator](#)
- [System.Reflection.Metadata.PropertyDefinitionHandleCollection](#)
- [System.Reflection.Metadata.ReservedBlob<THandle>](#)
- [System.Reflection.Metadata.SequencePoint](#)
- [System.Reflection.Metadata.SequencePointCollection.Enumerator](#)
- [System.Reflection.Metadata.SequencePointCollection](#)
- [System.Reflection.Metadata.SignatureHeader](#)
- [System.Reflection.Metadata.StandaloneSignature](#)
- [System.Reflection.Metadata.StandaloneSignatureHandle](#)

- [System.Reflection.Metadata.StringHandle](#)
- [System.Reflection.Metadata.TypeDefinition](#)
- [System.Reflection.Metadata.TypeDefinitionHandle](#)
- [System.Reflection.Metadata.TypeDefinitionHandleCollection.Enumerator](#)
- [System.Reflection.Metadata.TypeDefinitionHandleCollection](#)
- [System.Reflection.Metadata.TypeLayout](#)
- [System.Reflection.Metadata.TypeReference](#)
- [System.Reflection.Metadata.TypeReferenceHandle](#)
- [System.Reflection.Metadata.TypeReferenceHandleCollection.Enumerator](#)
- [System.Reflection.Metadata.TypeReferenceHandleCollection](#)
- [System.Reflection.Metadata.TypeSpecification](#)
- [System.Reflection.Metadata.TypeSpecificationHandle](#)
- [System.Reflection.Metadata.UserStringHandle](#)
- [System.Reflection.ParameterModifier](#)
- [System.Reflection.PortableExecutable.CodeViewDebugDirectoryData](#)
- [System.Reflection.PortableExecutable.DebugDirectoryEntry](#)
- [System.Reflection.PortableExecutable.PEMemoryBlock](#)
- [System.Reflection.PortableExecutable.SectionHeader](#)
- [System.Runtime.CompilerServices.AsyncTaskMethodBuilder<TResult>](#)
- [System.Runtime.CompilerServices.AsyncTaskMethodBuilder](#)
- [System.Runtime.CompilerServices.AsyncValueTaskMethodBuilder<TResult>](#)
- [System.Runtime.CompilerServices.AsyncValueTaskMethodBuilder](#)
- [System.Runtime.CompilerServices.AsyncVoidMethodBuilder](#)
- [System.Runtime.CompilerServices.ConfiguredTaskAwaitable<TResult>.ConfiguredTaskAwaiter](#)
- [System.Runtime.CompilerServices.ConfiguredTaskAwaitable<TResult>](#)
- [System.Runtime.CompilerServices.ConfiguredTaskAwaitable.ConfiguredTaskAwaiter](#)
- [System.Runtime.CompilerServices.ConfiguredTaskAwaitable](#)
- [System.Runtime.CompilerServices.ConfiguredValueTaskAwaitable<TResult>](#)
- [System.Runtime.CompilerServices.ConfiguredValueTaskAwaitable<TResult>.ConfiguredValueTaskAwaiter](#)
- [System.Runtime.CompilerServices.TaskAwaiter<TResult>](#)
- [System.Runtime.CompilerServices.TaskAwaiter](#)
- [System.Runtime.CompilerServices.ValueTaskAwaiter<TResult>](#)
- [System.Runtime.CompilerServices.ValueTaskAwaiter<TResult>](#)
- [System.Runtime.InteropServices.ArrayWithOffset](#)
- [System.Runtime.InteropServices.GCHandle](#)
- [System.Runtime.InteropServices.HandleRef](#)
- [System.Runtime.InteropServices.OSPlatform](#)
- [System.Runtime.InteropServices.WindowsRuntime.EventRegistrationToken](#)

- [System.Runtime.Serialization.SerializationEntry](#)
- [System.Runtime.Serialization.StreamingContext](#)
- [System.RuntimeArgumentHandle](#)
- [System.RuntimeFieldHandle](#)
- [System.RuntimeMethodHandle](#)
- [System.RuntimeTypeHandle](#)
- [System.SByte](#)
- [System.Security.Cryptography.CngProperty](#)
- [System.Security.Cryptography.ECCurve](#)
- [System.Security.Cryptography.HashAlgorithmName](#)
- [System.Security.Cryptography.X509Certificates.X509ChainStatus](#)
- [System.Security.Cryptography.Xml.X509IssuerSerial](#)
- [System.ServiceProcess.SessionChangeDescription](#)
- [System.Single](#)
- [System.Span<T>.Enumerator](#)
- [System.Span<T>](#)
- [System.Threading.AsyncFlowControl](#)
- [System.Threading.AsyncLocalValueChangedArgs<T>](#)
- [System.Threading.CancellationToken](#)
- [System.Threading.CancellationTokenRegistration](#)
- [System.Threading.LockCookie](#)
- [System.Threading.SpinLock](#)
- [System.Threading.SpinWait](#)
- [System.Threading.Tasks.Dataflow.DataflowMessageHeader](#)
- [System.Threading.Tasks.ParallelLoopResult](#)
- [System.Threading.Tasks.ValueTask<TResult>](#)
- [System.TimeSpan](#)
- [System.TimeZoneInfo.TransitionTime](#)
- [System.Transactions.TransactionOptions](#)
- [System.TypedReference](#)
- [System.TypedReference](#)
- [System.UInt16](#)
- [System.UInt32](#)
- [System.UInt64](#)
- [System.UIntPtr](#)
- [System.Windows.Forms.ColorDialog.Color](#)
- [System.Windows.Media.Animation.KeyTime](#)
- [System.Windows.Media.Animation.RepeatBehavior](#)
- [System.Xml.Serialization.XmlDeserializationEvents](#)
- [Windows.Foundation.Point](#)

- [Windows.Foundation.Rect](#)
 - [Windows.Foundation.Size](#)
 - [Windows.UI.Color](#)
 - [Windows.UI.Xaml.Controls.Primitives.GeneratorPosition](#)
 - [Windows.UI.Xaml.CornerRadius](#)
 - [Windows.UI.Xaml.Duration](#)
 - [Windows.UI.Xaml.GridLength](#)
 - [Windows.UI.Xaml.Media.Matrix](#)
 - [Windows.UI.Xaml.Media.Media3D.Matrix3D](#)
 - [Windows.UI.Xaml.Thickness](#)
-

OpenSSL versions on macOS

The .NET Core 3.0 and later runtimes on macOS now prefer OpenSSL 1.1.x versions to OpenSSL 1.0.x versions for the [AesCcm](#), [AesGcm](#), [DSAOpenSsl](#), [ECDiffieHellmanOpenSsl](#), [ECDsaOpenSsl](#), [RSAOpenSsl](#), and [SafeEvpPKeyHandle](#) types.

The .NET Core 2.1 runtime now supports OpenSSL 1.1.x versions, but still prefers OpenSSL 1.0.x versions.

Change description

Previously, the .NET Core runtime used OpenSSL 1.0.x versions on macOS for types that interact with OpenSSL. The most recent OpenSSL 1.0.x version, OpenSSL 1.0.2, is now out of support. To keep types that use OpenSSL on supported versions of OpenSSL, the .NET Core 3.0 and later runtimes now use newer versions of OpenSSL on macOS.

With this change, the behavior for the .NET Core runtimes on macOS is as follows:

- The .NET Core 3.0 and later version runtimes use OpenSSL 1.1.x, if available, and fall back to OpenSSL 1.0.x only if there's no 1.1.x version available.

For callers that use the OpenSSL interop types with custom P/Invokes, follow the guidance in the [SafeEvpPKeyHandle.OpenSslVersion](#) remarks. Your app may crash if you don't check the [OpenSslVersion](#) value.

- The .NET Core 2.1 runtime uses OpenSSL 1.0.x, if available, and falls back to OpenSSL 1.1.x if there's no 1.0.x version available.

The 2.1 runtime prefers the earlier version of OpenSSL because the [SafeEvpPKeyHandle.OpenSslVersion](#) property does not exist in .NET Core 2.1, so the OpenSSL version cannot be reliably determined at run time.

Version introduced

- .NET Core 2.1.16
- .NET Core 3.0.3
- .NET Core 3.1.2

Recommended action

- Uninstall OpenSSL version 1.0.2 if it's no longer needed.
- Install OpenSSL 1.1.x if you use the [AesCcm](#), [AesGcm](#), [DSAOpenSsl](#), [ECDiffieHellmanOpenSsl](#), [ECDsaOpenSsl](#), [RSAOpenSsl](#), or [SafeEvpPKeyHandle](#) types.
- If you use the OpenSSL interop types with custom P/Invokes, follow the guidance in the [SafeEvpPKeyHandle.OpenSslVersion](#) remarks.

Category

Core .NET libraries

Affected APIs

- [System.Security.Cryptography.AesCcm](#)
- [System.Security.Cryptography.AesGcm](#)
- [System.Security.Cryptography.DSAOpenSsl](#)
- [System.Security.Cryptography.ECDiffieHellmanOpenSsl](#)
- [System.Security.Cryptography.ECDsaOpenSsl](#)
- [System.Security.Cryptography.RSAOpenSsl](#)
- [System.Security.Cryptography.SafeEvpPKeyHandle](#)

MSBuild

- [Project tools now included in SDK](#)

Project tools now included in SDK

The .NET Core 2.1 SDK now includes common CLI tooling, and you no longer need to reference these tools from the project.

Change description

In .NET Core 2.0, projects reference external .NET tools with the `<DotNetCliToolReference>` project setting. In .NET Core 2.1, some of these tools are included with the .NET Core SDK, and the setting is no longer needed. If you include references to these tools in your project, you'll receive an error similar to the following:
The tool 'Microsoft.EntityFrameworkCore.Tools.DotNet' is now included in the .NET Core SDK.

Tools now included in .NET Core 2.1 SDK:

| <code><DotNetCliToolReference> value</code> | Tool |
|---|------------------------------------|
| <code>Microsoft.DotNet.Watcher.Tools</code> | <code>dotnet-watch</code> |
| <code>Microsoft.Extensions.SecretManager.Tools</code> | <code>dotnet-user-secrets</code> ↗ |
| <code>Microsoft.Extensions.Caching.SqlConfig.Tools</code> | <code>dotnet-sql-cache</code> ↗ |
| <code>Microsoft.EntityFrameworkCore.Tools.DotNet</code> | <code>dotnet-ef</code> |

Version introduced

.NET Core SDK 2.1.300

Recommended action

Remove the `<DotNetCliToolReference>` setting from your project.

Category

MSBuild

Affected APIs

N/A

See also

- [What's new in .NET Core 2.1](#)

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

What's new in .NET Core 2.0

Article • 02/18/2022

.NET Core 2.0 includes enhancements and new features in the following areas:

- [Tooling](#)
- [Language support](#)
- [Platform improvements](#)
- [API changes](#)
- [Visual Studio integration](#)
- [Documentation improvements](#)

Tooling

dotnet restore runs implicitly

In previous versions of .NET Core, you had to run the [dotnet restore](#) command to download dependencies immediately after you created a new project with the [dotnet new](#) command, as well as whenever you added a new dependency to your project.

You don't have to run [dotnet restore](#) because it's run implicitly by all commands that require a restore to occur, such as `dotnet new`, `dotnet build`, `dotnet run`, `dotnet test`, `dotnet publish`, and `dotnet pack`. To disable implicit restore, use the `--no-restore` option.

The `dotnet restore` command is still useful in certain scenarios where explicitly restoring makes sense, such as [continuous integration builds in Azure DevOps Services](#) or in build systems that need to explicitly control when the restore occurs.

For information about how to manage NuGet feeds, see the [dotnet restore documentation](#).

You can also disable the automatic invocation of `dotnet restore` by passing the `--no-restore` switch to the `new`, `run`, `build`, `publish`, `pack`, and `test` commands.

Retargeting to .NET Core 2.0

If the .NET Core 2.0 SDK is installed, projects that target .NET Core 1.x can be retargeted to .NET Core 2.0.

To retarget to .NET Core 2.0, edit your project file by changing the value of the `<TargetFramework>` element (or the `<TargetFrameworks>` element if you have more than one target in your project file) from 1.x to 2.0:

XML

```
<PropertyGroup>
  <TargetFramework>netcoreapp2.0</TargetFramework>
</PropertyGroup>
```

You can also retarget .NET Standard libraries to .NET Standard 2.0 the same way:

XML

```
<PropertyGroup>
  <TargetFramework>netstandard2.0</TargetFramework>
</PropertyGroup>
```

For more information about migrating your project to .NET Core 2.0, see [Migrating from ASP.NET Core 1.x to ASP.NET Core 2.0](#).

Language support

In addition to supporting C# and F#, .NET Core 2.0 also supports Visual Basic.

Visual Basic

With version 2.0, .NET Core now supports Visual Basic 2017. You can use Visual Basic to create the following project types:

- .NET Core console apps
- .NET Core class libraries
- .NET Standard class libraries
- .NET Core unit test projects
- .NET Core xUnit test projects

For example, to create a Visual Basic "Hello World" application, do the following steps from the command line:

1. Open a console window, create a directory for your project, and make it the current directory.
2. Enter the command `dotnet new console -lang vb`.

The command creates a project file with a `.vbproj` file extension, along with a Visual Basic source code file named `Program.vb`. This file contains the source code to write the string "Hello World!" to the console window.

3. Enter the command `dotnet run`. The [.NET Core CLI](#) automatically compiles and executes the application, which displays the message "Hello World!" in the console window.

Support for C# 7.1

.NET Core 2.0 supports C# 7.1, which adds a number of new features, including:

- The `Main` method, the application entry point, can be marked with the `async` keyword.
- Inferred tuple names.
- Default expressions.

Platform improvements

.NET Core 2.0 includes a number of features that make it easier to install .NET Core and to use it on supported operating systems.

.NET Core for Linux is a single implementation

.NET Core 2.0 offers a single Linux implementation that works on multiple Linux distributions. .NET Core 1.x required that you download a distribution-specific Linux implementation.

You can also develop apps that target Linux as a single operating system. .NET Core 1.x required that you target each Linux distribution separately.

Support for the Apple cryptographic libraries

.NET Core 1.x on macOS required the OpenSSL toolkit's cryptographic library. .NET Core 2.0 uses the Apple cryptographic libraries and doesn't require OpenSSL, so you no longer need to install it.

API changes and library support

Support for .NET Standard 2.0

.NET Standard defines a versioned set of APIs that must be available on .NET implementations that comply with that version of the standard. .NET Standard is targeted at library developers. It aims to guarantee the functionality that is available to a library that targets a version of .NET Standard on each .NET implementation. .NET Core 1.x supports .NET Standard version 1.6; .NET Core 2.0 supports the latest version, .NET Standard 2.0. For more information, see [.NET Standard](#).

.NET Standard 2.0 includes over 20,000 more APIs than were available in .NET Standard 1.6. Much of this expanded surface area results from incorporating APIs that are common to the .NET Framework and Xamarin into .NET Standard.

.NET Standard 2.0 class libraries can also reference .NET Framework class libraries, provided that they call APIs that are present in .NET Standard 2.0. No recompilation of the .NET Framework libraries is required.

For a list of the APIs that have been added to .NET Standard since its last version, .NET Standard 1.6, see [.NET Standard 2.0 vs. 1.6](#).

Expanded surface area

The total number of APIs available on .NET Core 2.0 has more than doubled in comparison with .NET Core 1.1.

And with the [Windows Compatibility Pack](#) porting from .NET Framework has also become much simpler.

Support for .NET Framework libraries

.NET Core code can reference existing .NET Framework libraries, including existing NuGet packages. Note that the libraries must use APIs that are found in .NET Standard.

Visual Studio integration

Visual Studio 2017 version 15.3 and in some cases Visual Studio for Mac offer a number of significant enhancements for .NET Core developers.

Retargeting .NET Core apps and .NET Standard libraries

If the .NET Core 2.0 SDK is installed, you can retarget .NET Core 1.x projects to .NET Core 2.0 and .NET Standard 1.x libraries to .NET Standard 2.0.

To retarget your project in Visual Studio, you open the **Application** tab of the project's properties dialog and change the **Target framework** value to **.NET Core 2.0** or **.NET Standard 2.0**. You can also change it by right-clicking on the project and selecting the **Edit *.csproj file** option. For more information, see the [Tooling](#) section earlier in this topic.

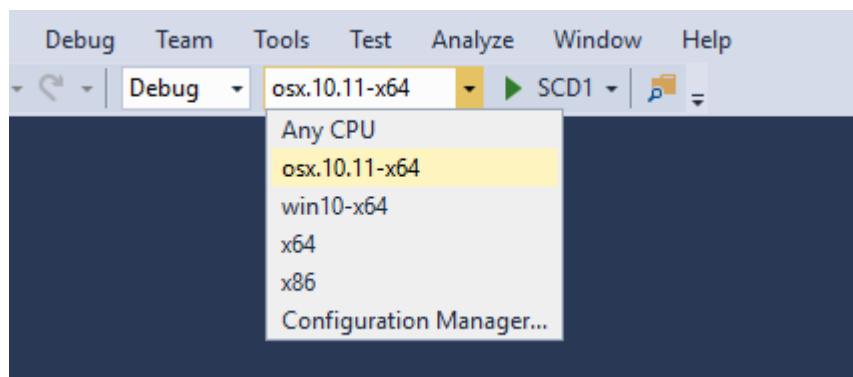
Live Unit Testing support for .NET Core

Whenever you modify your code, Live Unit Testing automatically runs any affected unit tests in the background and displays the results and code coverage live in the Visual Studio environment. .NET Core 2.0 now supports Live Unit Testing. Previously, Live Unit Testing was available only for .NET Framework applications.

For more information, see [Live Unit Testing with Visual Studio](#) and the [Live Unit Testing FAQ](#).

Better support for multiple target frameworks

If you're building a project for multiple target frameworks, you can now select the target platform from the top-level menu. In the following figure, a project named SCD1 targets 64-bit macOS X 10.11 (osx.10.11-x64) and 64-bit Windows 10/Windows Server 2016 (win10-x64). You can select the target framework before selecting the project button, in this case to run a debug build.



Side-by-side support for .NET Core SDKs

You can now install the .NET Core SDK independently of Visual Studio. This makes it possible for a single version of Visual Studio to build projects that target different versions of .NET Core. Previously, Visual Studio and the .NET Core SDK were tightly coupled; a particular version of the SDK accompanied a particular version of Visual Studio.

Documentation improvements

.NET Application Architecture

[.NET Application Architecture](#) ↗ gives you access to a set of e-books that provide guidance, best practices, and sample applications when using .NET to build:

- Microservices and Docker containers
- Web applications with ASP.NET
- Mobile applications with Xamarin
- Applications that are deployed to the Cloud with Azure

See also

- [What's new in ASP.NET Core 2.0](#)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

 .NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

What's new in .NET Standard

Article • 10/11/2022

.NET Standard is a formal specification that defines a versioned set of APIs that must be available on .NET implementations that comply with that version of the standard. .NET Standard is targeted at library developers. A library that targets a .NET Standard version can be used on any .NET or Xamarin implementation that supports that version of the standard.

.NET Standard is included with the .NET SDK. It's also included with Visual Studio if you select the .NET workload.

.NET Standard 2.1 is the last version of .NET Standard that will be released. For more information, see [.NET 5+ and .NET Standard](#).

Supported .NET implementations

.NET Standard 2.1 is supported by the following .NET implementations:

- .NET Core 3.0 or later (including .NET 5 and later)
- Mono 6.4 or later
- Xamarin.iOS 12.16 or later
- Xamarin.Android 10.0 or later

.NET Standard 2.0 is supported by the following .NET implementations:

- .NET Core 2.0 or later (including .NET 5 and later)
- .NET Framework 4.6.1 or later
- Mono 5.4 or later
- Xamarin.iOS 10.14 or later
- Xamarin.Mac 3.8 or later
- Xamarin.Android 8.0 or later
- Universal Windows Platform 10.0.16299 or later

What's new in .NET Standard 2.1

.NET Standard 2.1 adds many APIs to the standard. Some of them are new APIs, and others are existing APIs that help to converge the .NET implementations even further. For a list of the APIs that have been added to .NET Standard 2.1, see [.NET Standard 2.1 vs 2.0 ↗](#).

For more information, see the [Announcing .NET Standard 2.1](#) blog post.

What's new in .NET Standard 2.0

.NET Standard 2.0 includes the following new features.

A vastly expanded set of APIs

Through version 1.6, .NET Standard included a comparatively small subset of APIs. Among those excluded were many APIs that were commonly used in .NET Framework or Xamarin. This complicates development, since it requires that developers find suitable replacements for familiar APIs when they develop applications and libraries that target multiple .NET implementations. .NET Standard 2.0 addresses this limitation by adding over 20,000 more APIs than were available in .NET Standard 1.6, the previous version of the standard. For a list of the APIs that have been added to .NET Standard 2.0, see [.NET Standard 2.0 vs 1.6](#).

Some of the additions to the [System](#) namespace in .NET Standard 2.0 include:

- Support for the [AppDomain](#) class.
- Better support for working with arrays from additional members in the [Array](#) class.
- Better support for working with attributes from additional members in the [Attribute](#) class.
- Better calendar support and additional formatting options for [DateTime](#) values.
- Additional [Decimal](#) rounding functionality.
- Additional functionality in the [Environment](#) class.
- Enhanced control over the garbage collector through the [GC](#) class.
- Enhanced support for string comparison, enumeration, and normalization in the [String](#) class.
- Support for daylight saving adjustments and transition times in the [TimeZoneInfo.AdjustmentRule](#) and [TimeZoneInfo.TransitionTime](#) classes.
- Significantly enhanced functionality in the [Type](#) class.
- Better support for deserialization of exception objects by adding an exception constructor with [SerializationInfo](#) and [StreamingContext](#) parameters.

Support for .NET Framework libraries

Many libraries target .NET Framework rather than .NET Standard. However, most of the calls in those libraries are to APIs that are included in .NET Standard 2.0. Starting with .NET Standard 2.0, you can access .NET Framework libraries from a .NET Standard library

by using a [compatibility shim](#). This compatibility layer is transparent to developers; you don't have to do anything to take advantage of .NET Framework libraries.

The single requirement is that the APIs called by the .NET Framework class library must be included in .NET Standard 2.0.

Support for Visual Basic

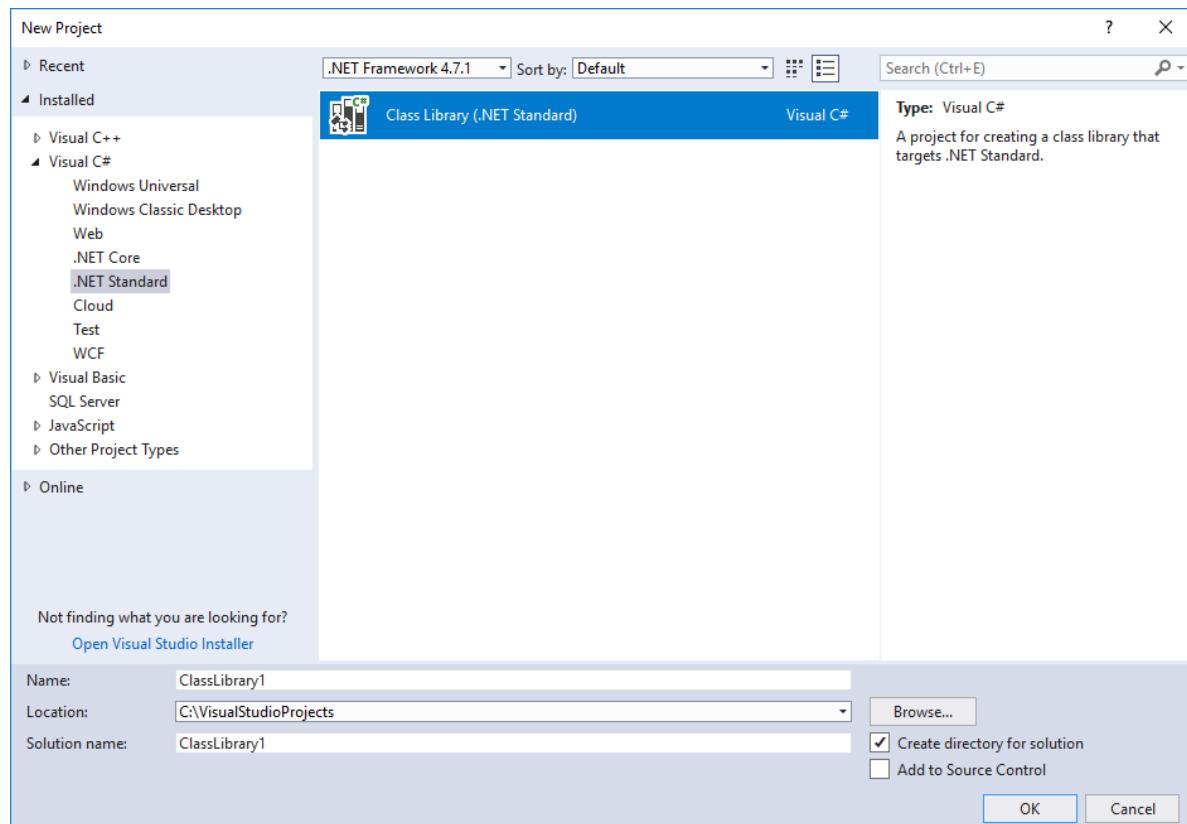
You can now develop .NET Standard libraries in Visual Basic. Visual Studio 2019 and Visual Studio 2017 version 15.3 or later with the .NET Core workload installed include a .NET Standard Class Library template. For Visual Basic developers who use other development tools and environments, you can use the [dotnet new](#) command to create a .NET Standard Library project. For more information, see the [Tooling support for .NET Standard libraries](#).

Tooling support for .NET Standard libraries

With the release of .NET Core 2.0 and .NET Standard 2.0, both Visual Studio 2017 and the [.NET CLI](#) include tooling support for creating .NET Standard libraries.

If you install Visual Studio with the [.NET Core cross-platform development](#) workload, you can create a .NET Standard 2.0 library project by using a project template, as the following figure shows:





If you're using the .NET CLI, the following `dotnet new` command creates a class library project that targets .NET Standard 2.0:

```
.NET CLI  
  
dotnet new classlib
```

See also

- [.NET Standard](#)
- [Introducing .NET Standard ↗](#)
- [Download the .NET SDK ↗](#)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For



.NET feedback

The .NET documentation is open source. Provide feedback here.

[Open a documentation issue](#)

[Provide product feedback](#)

more information, see [our contributor guide](#).

Common type system

Article • 01/03/2024

The common type system defines how types are declared, used, and managed in the common language runtime, and is also an important part of the runtime's support for cross-language integration. The common type system performs the following functions:

- Establishes a framework that helps enable cross-language integration, type safety, and high-performance code execution.
- Provides an object-oriented model that supports the complete implementation of many programming languages.
- Defines rules that languages must follow, which helps ensure that objects written in different languages can interact with each other.
- Provides a library that contains the primitive data types (such as [Boolean](#), [Byte](#), [Char](#), [Int32](#), and [UInt64](#)) used in application development.

Types in .NET

All types in .NET are either value types or reference types.

Value types are data types whose objects are represented by the object's actual value. If an instance of a value type is assigned to a variable, that variable is given a fresh copy of the value.

Reference types are data types whose objects are represented by a reference (similar to a pointer) to the object's actual value. If a reference type is assigned to a variable, that variable references (points to) the original value. No copy is made.

The common type system in .NET supports the following five categories of types:

- [Classes](#)
- [Structures](#)
- [Enumerations](#)
- [Interfaces](#)
- [Delegates](#)

Classes

A class is a reference type that can be derived directly from another class and that is derived implicitly from [System.Object](#). The class defines the operations that an object (which is an instance of the class) can perform (methods, events, or properties) and the data that the object contains (fields). Although a class generally includes both definition and implementation (unlike interfaces, for example, which contain only definition without implementation), it can have one or more members that have no implementation.

The following table describes some of the characteristics that a class may have. Each language that supports the runtime provides a way to indicate that a class or class member has one or more of these characteristics. However, individual programming languages that target .NET may not make all these characteristics available.

[+] [Expand table](#)

| Characteristic | Description |
|--------------------------|--|
| sealed | Specifies that another class cannot be derived from this type. |
| implements | Indicates that the class uses one or more interfaces by providing implementations of interface members. |
| abstract | Indicates that the class cannot be instantiated. To use it, you must derive another class from it. |
| inherits | Indicates that instances of the class can be used anywhere the base class is specified. A derived class that inherits from a base class can use the implementation of any public members provided by the base class, or the derived class can override the implementation of the public members with its own implementation. |
| exported or not exported | Indicates whether a class is visible outside the assembly in which it is defined. This characteristic applies only to top-level classes and not to nested classes. |

Note

A class can also be nested in a parent class or structure. Nested classes also have member characteristics. For more information, see [Nested Types](#).

Class members that have no implementation are abstract members. A class that has one or more abstract members is itself abstract; new instances of it cannot be created. Some languages that target the runtime let you mark a class as abstract even if none of its members are abstract. You can use an abstract class when you want to encapsulate a

basic set of functionality that derived classes can inherit or override when appropriate. Classes that are not abstract are referred to as concrete classes.

A class can implement any number of interfaces, but it can inherit from only one base class in addition to [System.Object](#), from which all classes inherit implicitly. All classes must have at least one constructor, which initializes new instances of the class. If you do not explicitly define a constructor, most compilers will automatically provide a parameterless constructor.

Structures

A structure is a value type that derives implicitly from [System.ValueType](#), which in turn is derived from [System.Object](#). A structure is useful for representing values whose memory requirements are small, and for passing values as by-value parameters to methods that have strongly typed parameters. In .NET, all primitive data types ([Boolean](#), [Byte](#), [Char](#), [DateTime](#), [Decimal](#), [Double](#), [Int16](#), [Int32](#), [Int64](#), [SByte](#), [Single](#), [UInt16](#), [UInt32](#), and [UInt64](#)) are defined as structures.

Like classes, structures define both data (the fields of the structure) and the operations that can be performed on that data (the methods of the structure). This means that you can call methods on structures, including the virtual methods defined on the [System.Object](#) and [System.ValueType](#) classes, and any methods defined on the value type itself. In other words, structures can have fields, properties, and events, as well as static and nonstatic methods. You can create instances of structures, pass them as parameters, store them as local variables, or store them in a field of another value type or reference type. Structures can also implement interfaces.

Value types also differ from classes in several respects. First, although they implicitly inherit from [System.ValueType](#), they cannot directly inherit from any type. Similarly, all value types are sealed, which means that no other type can be derived from them. They also do not require constructors.

For each value type, the common language runtime supplies a corresponding boxed type, which is a class that has the same state and behavior as the value type. An instance of a value type is boxed when it is passed to a method that accepts a parameter of type [System.Object](#). It is unboxed (that is, converted from an instance of a class back to an instance of a value type) when control returns from a method call that accepts a value type as a by-reference parameter. Some languages require that you use special syntax when the boxed type is required; others automatically use the boxed type when it is needed. When you define a value type, you are defining both the boxed and the unboxed type.

Enumerations

An enumeration is a value type that inherits directly from [System.Enum](#) and that supplies alternate names for the values of an underlying primitive type. An enumeration type has a name, an underlying type that must be one of the built-in signed or unsigned integer types (such as [Byte](#), [Int32](#), or [UInt64](#)), and a set of fields. The fields are static literal fields, each of which represents a constant. The same value can be assigned to multiple fields. When this occurs, you must mark one of the values as the primary enumeration value for reflection and string conversion.

You can assign a value of the underlying type to an enumeration and vice versa (no cast is required by the runtime). You can create an instance of an enumeration and call the methods of [System.Enum](#), as well as any methods defined on the enumeration's underlying type. However, some languages might not let you pass an enumeration as a parameter when an instance of the underlying type is required (or vice versa).

The following additional restrictions apply to enumerations:

- They cannot define their own methods.
- They cannot implement interfaces.
- They cannot define properties or events.
- They cannot be generic, unless they are generic only because they are nested within a generic type. That is, an enumeration cannot have type parameters of its own.

(!) Note

Nested types (including enumerations) created with Visual Basic, C#, and C++ include the type parameters of all enclosing generic types, and are therefore generic even if they do not have type parameters of their own. For more information, see "Nested Types" in the [Type.MakeGenericType](#) reference topic.

The [FlagsAttribute](#) attribute denotes a special kind of enumeration called a bit field. The runtime itself does not distinguish between traditional enumerations and bit fields, but your language might do so. When this distinction is made, bitwise operators can be used on bit fields, but not on enumerations, to generate unnamed values. Enumerations are generally used for lists of unique elements, such as days of the week, country or region names, and so on. Bit fields are generally used for lists of qualities or quantities that might occur in combination, such as `Red And Big And Fast`.

The following example shows how to use both bit fields and traditional enumerations.

C#

```
using System;
using System.Collections.Generic;

// A traditional enumeration of some root vegetables.
public enum SomeRootVegetables
{
    HorseRadish,
    Radish,
    Turnip
}

// A bit field or flag enumeration of harvesting seasons.
[Flags]
public enum Seasons
{
    None = 0,
    Summer = 1,
    Autumn = 2,
    Winter = 4,
    Spring = 8,
    All = Summer | Autumn | Winter | Spring
}

public class Example
{
    public static void Main()
    {
        // Hash table of when vegetables are available.
        Dictionary<SomeRootVegetables, Seasons> AvailableIn = new
        Dictionary<SomeRootVegetables, Seasons>();

        AvailableIn[SomeRootVegetables.HorseRadish] = Seasons.All;
        AvailableIn[SomeRootVegetables.Radish] = Seasons.Spring;
        AvailableIn[SomeRootVegetables.Turnip] = Seasons.Spring |
            Seasons.Autumn;

        // Array of the seasons, using the enumeration.
        Seasons[] theSeasons = new Seasons[] { Seasons.Summer,
        Seasons.Autumn,
        Seasons.Winter, Seasons.Spring };

        // Print information of what vegetables are available each season.
        foreach (Seasons season in theSeasons)
        {
            Console.Write(String.Format(
                "The following root vegetables are harvested in {0}:\n",
                season.ToString("G")));
            foreach (KeyValuePair<SomeRootVegetables, Seasons> item in
AvailableIn)
            {

```

```

        // A bitwise comparison.
        if (((Seasons)item.Value & season) > 0)
            Console.WriteLine(String.Format(" {0:G}\n",
                (SomeRootVegetables)item.Key));
    }
}
}

// The example displays the following output:
//   The following root vegetables are harvested in Summer:
//     HorseRadish
//   The following root vegetables are harvested in Autumn:
//     Turnip
//     HorseRadish
//   The following root vegetables are harvested in Winter:
//     HorseRadish
//   The following root vegetables are harvested in Spring:
//     Turnip
//     Radish
//     HorseRadish

```

Interfaces

An interface defines a contract that specifies a "can do" relationship or a "has a" relationship. Interfaces are often used to implement functionality, such as comparing and sorting (the [IComparable](#) and [IComparable<T>](#) interfaces), testing for equality (the [IEquatable<T>](#) interface), or enumerating items in a collection (the [IEnumerable](#) and [IEnumerator<T>](#) interfaces). Interfaces can have properties, methods, and events, all of which are abstract members; that is, although the interface defines the members and their signatures, it leaves it to the type that implements the interface to define the functionality of each interface member. This means that any class or structure that implements an interface must supply definitions for the abstract members declared in the interface. An interface can require any implementing class or structure to also implement one or more other interfaces.

The following restrictions apply to interfaces:

- An interface can be declared with any accessibility, but interface members must all have public accessibility.
- Interfaces cannot define constructors.
- Interfaces cannot define fields.
- Interfaces can define only instance members. They cannot define static members.

Each language must provide rules for mapping an implementation to the interface that requires the member, because more than one interface can declare a member with the same signature, and these members can have separate implementations.

Delegates

Delegates are reference types that serve a purpose similar to that of function pointers in C++. They are used for event handlers and callback functions in .NET. Unlike function pointers, delegates are secure, verifiable, and type safe. A delegate type can represent any instance method or static method that has a compatible signature.

A parameter of a delegate is compatible with the corresponding parameter of a method if the type of the delegate parameter is more restrictive than the type of the method parameter, because this guarantees that an argument passed to the delegate can be passed safely to the method.

Similarly, the return type of a delegate is compatible with the return type of a method if the return type of the method is more restrictive than the return type of the delegate, because this guarantees that the return value of the method can be cast safely to the return type of the delegate.

For example, a delegate that has a parameter of type [IEnumerable](#) and a return type of [Object](#) can represent a method that has a parameter of type [Object](#) and a return value of type [IEnumerable](#). For more information and example code, see [Delegate.CreateDelegate\(Type, Object, MethodInfo\)](#).

A delegate is said to be bound to the method it represents. In addition to being bound to the method, a delegate can be bound to an object. The object represents the first parameter of the method, and is passed to the method every time the delegate is invoked. If the method is an instance method, the bound object is passed as the implicit `this` parameter (`Me` in Visual Basic); if the method is static, the object is passed as the first formal parameter of the method, and the delegate signature must match the remaining parameters. For more information and example code, see [System.Delegate](#).

All delegates inherit from [System.MulticastDelegate](#), which inherits from [System.Delegate](#). The C#, Visual Basic, and C++ languages do not allow inheritance from these types. Instead, they provide keywords for declaring delegates.

Because delegates inherit from [MulticastDelegate](#), a delegate has an invocation list, which is a list of methods that the delegate represents and that are executed when the delegate is invoked. All methods in the list receive the arguments supplied when the delegate is invoked.

ⓘ Note

The return value is not defined for a delegate that has more than one method in its invocation list, even if the delegate has a return type.

In many cases, such as with callback methods, a delegate represents only one method, and the only actions you have to take are creating the delegate and invoking it.

For delegates that represent multiple methods, .NET provides methods of the [Delegate](#) and [MulticastDelegate](#) delegate classes to support operations such as adding a method to a delegate's invocation list (the [Delegate.Combine](#) method), removing a method (the [Delegate.Remove](#) method), and getting the invocation list (the [Delegate.GetInvocationList](#) method).

ⓘ Note

It is not necessary to use these methods for event-handler delegates in C#, C++, and Visual Basic, because these languages provide syntax for adding and removing event handlers.

Type definitions

A type definition includes the following:

- Any attributes defined on the type.
- The type's accessibility (visibility).
- The type's name.
- The type's base type.
- Any interfaces implemented by the type.
- Definitions for each of the type's members.

Attributes

Attributes provide additional user-defined metadata. Most commonly, they are used to store additional information about a type in its assembly, or to modify the behavior of a type member in either the design-time or run-time environment.

Attributes are themselves classes that inherit from [System.Attribute](#). Languages that support the use of attributes each have their own syntax for applying attributes to a language element. Attributes can be applied to almost any language element; the specific elements to which an attribute can be applied are defined by the [AttributeUsageAttribute](#) that is applied to that attribute class.

Type accessibility

All types have a modifier that governs their accessibility from other types. The following table describes the type accessibilities supported by the runtime.

[+] Expand table

| Accessibility | Description |
|---------------|---|
| public | The type is accessible by all assemblies. |
| assembly | The type is accessible only from within its assembly. |

The accessibility of a nested type depends on its accessibility domain, which is determined by both the declared accessibility of the member and the accessibility domain of the immediately containing type. However, the accessibility domain of a nested type cannot exceed that of the containing type.

The accessibility domain of a nested member `M` declared in a type `T` within a program `P` is defined as follows (noting that `M` might itself be a type):

- If the declared accessibility of `M` is `public`, the accessibility domain of `M` is the accessibility domain of `T`.
- If the declared accessibility of `M` is `protected internal`, the accessibility domain of `M` is the intersection of the accessibility domain of `T` with the program text of `P` and the program text of any type derived from `T` declared outside `P`.
- If the declared accessibility of `M` is `protected`, the accessibility domain of `M` is the intersection of the accessibility domain of `T` with the program text of `T` and any type derived from `T`.
- If the declared accessibility of `M` is `internal`, the accessibility domain of `M` is the intersection of the accessibility domain of `T` with the program text of `P`.
- If the declared accessibility of `M` is `private`, the accessibility domain of `M` is the program text of `T`.

Type Names

The common type system imposes only two restrictions on names:

- All names are encoded as strings of Unicode (16-bit) characters.

- Names are not permitted to have an embedded (16-bit) value of 0x0000.

However, most languages impose additional restrictions on type names. All comparisons are done on a byte-by-byte basis, and are therefore case-sensitive and locale-independent.

Although a type might reference types from other modules and assemblies, a type must be fully defined within one .NET module. (Depending on compiler support, however, it can be divided into multiple source code files.) Type names need be unique only within a namespace. To fully identify a type, the type name must be qualified by the namespace that contains the implementation of the type.

Base types and interfaces

A type can inherit values and behaviors from another type. The common type system does not allow types to inherit from more than one base type.

A type can implement any number of interfaces. To implement an interface, a type must implement all the virtual members of that interface. A virtual method can be implemented by a derived type and can be invoked either statically or dynamically.

Type members

The runtime enables you to define members of your type, which specifies the behavior and state of a type. Type members include the following:

- [Fields](#)
- [Properties](#)
- [Methods](#)
- [Constructors](#)
- [Events](#)
- [Nested types](#)

Fields

A field describes and contains part of the type's state. Fields can be of any type supported by the runtime. Most commonly, fields are either `private` or `protected`, so that they are accessible only from within the class or from a derived class. If the value of a field can be modified from outside its type, a property set accessor is typically used. Publicly exposed fields are usually read-only and can be of two types:

- Constants, whose value is assigned at design time. These are static members of a class, although they are not defined using the `static` (`Shared` in Visual Basic) keyword.
- Read-only variables, whose values can be assigned in the class constructor.

The following example illustrates these two usages of read-only fields.

```
C#  
  
using System;  
  
public class Constants  
{  
    public const double Pi = 3.1416;  
    public readonly DateTime BirthDate;  
  
    public Constants(DateTime birthDate)  
    {  
        this.BirthDate = birthDate;  
    }  
}  
  
public class Example  
{  
    public static void Main()  
    {  
        Constants con = new Constants(new DateTime(1974, 8, 18));  
        Console.WriteLine(Constants.Pi + "\n");  
        Console.WriteLine(con.BirthDate.ToString("d") + "\n");  
    }  
}  
// The example displays the following output if run on a system whose  
// current  
// culture is en-US:  
//     3.1416  
//     8/18/1974
```

Properties

A property names a value or state of the type and defines methods for getting or setting the property's value. Properties can be primitive types, collections of primitive types, user-defined types, or collections of user-defined types. Properties are often used to keep the public interface of a type independent from the type's actual representation. This enables properties to reflect values that are not directly stored in the class (for example, when a property returns a computed value) or to perform validation before values are assigned to private fields. The following example illustrates the latter pattern.

```
C#
```

```

using System;

public class Person
{
    private int m_Age;

    public int Age
    {
        get { return m_Age; }
        set {
            if (value < 0 || value > 125)
            {
                throw new ArgumentOutOfRangeException("The value of the Age
property must be between 0 and 125.");
            }
            else
            {
                m_Age = value;
            }
        }
    }
}

```

In addition to including the property itself, the Microsoft intermediate language (MSIL) for a type that contains a readable property includes a `get_propertyname` method, and the MSIL for a type that contains a writable property includes a `set_propertyname` method.

Methods

A method describes operations that are available on the type. A method's signature specifies the allowable types of all its parameters and of its return value.

Although most methods define the precise number of parameters required for method calls, some methods support a variable number of parameters. The final declared parameter of these methods is marked with the [ParamArrayAttribute](#) attribute.

Language compilers typically provide a keyword, such as `params` in C# and `ParamArray` in Visual Basic, that makes explicit use of [ParamArrayAttribute](#) unnecessary.

Constructors

A constructor is a special kind of method that creates new instances of a class or structure. Like any other method, a constructor can include parameters; however, constructors have no return value (that is, they return `void`).

If the source code for a class does not explicitly define a constructor, the compiler includes a parameterless constructor. However, if the source code for a class defines only parameterized constructors, the Visual Basic and C# compilers do not generate a parameterless constructor.

If the source code for a structure defines constructors, they must be parameterized; a structure cannot define a parameterless constructor, and compilers do not generate parameterless constructors for structures or other value types. All value types do have an implicit parameterless constructor. This constructor is implemented by the common language runtime and initializes all fields of the structure to their default values.

Events

An event defines an incident that can be responded to, and defines methods for subscribing to, unsubscribing from, and raising the event. Events are often used to inform other types of state changes. For more information, see [Events](#).

Nested types

A nested type is a type that is a member of some other type. Nested types should be tightly coupled to their containing type and must not be useful as a general-purpose type. Nested types are useful when the declaring type uses and creates instances of the nested type, and use of the nested type is not exposed in public members.

Nested types are confusing to some developers and should not be publicly visible unless there is a compelling reason for visibility. In a well-designed library, developers should rarely have to use nested types to instantiate objects or declare variables.

Characteristics of type members

The common type system allows type members to have a variety of characteristics; however, languages are not required to support all these characteristics. The following table describes member characteristics.

[] [Expand table](#)

| Characteristic | Can apply to | Description |
|----------------|---------------------------------|---|
| abstract | Methods, properties, and events | The type does not supply the method's implementation. Types that inherit or implement abstract methods must supply an implementation for the method. The only |

| Characteristic | Can apply to | Description |
|---|---|---|
| | | exception is when the derived type is itself an abstract type. All abstract methods are virtual. |
| private, family, assembly, family and assembly, family or assembly, or public | All | <p>Defines the accessibility of the member:</p> <p>private Accessible only from within the same type as the member, or within a nested type.</p> <p>family Accessible from within the same type as the member, and from derived types that inherit from it.</p> <p>assembly Accessible only in the assembly in which the type is defined.</p> <p>family and assembly Accessible only from types that qualify for both family and assembly access.</p> <p>family or assembly Accessible only from types that qualify for either family or assembly access.</p> <p>public Accessible from any type.</p> |
| final | Methods, properties, and events | The virtual method cannot be overridden in a derived type. |
| initialize-only | Fields | The value can only be initialized, and cannot be written after initialization. |
| instance | Fields, methods, properties, and events | If a member is not marked as <code>static</code> (C# and C++), <code>Shared</code> (Visual Basic), <code>virtual</code> (C# and C++), or <code>Overridable</code> (Visual Basic), it is an instance member (there is no <code>instance</code> keyword). There will be as many copies of such members in memory as there are objects that use it. |
| literal | Fields | The value assigned to the field is a fixed value, known at compile time, of a built-in value type. Literal fields are sometimes referred to as constants. |
| newslot or override | All | <p>Defines how the member interacts with inherited members that have the same signature:</p> <p>newslot</p> |

| Characteristic | Can apply to | Description |
|----------------|--|---|
| | | Hides inherited members that have the same signature. |
| | override | Replaces the definition of an inherited virtual method. |
| | | The default is newslot. |
| static | Fields, methods, properties, and events | The member belongs to the type it is defined on, not to a particular instance of the type; the member exists even if an instance of the type is not created, and it is shared among all instances of the type. |
| virtual | Methods, properties, and events | The method can be implemented by a derived type and can be invoked either statically or dynamically. If dynamic invocation is used, the type of the instance that makes the call at run time (rather than the type known at compile time) determines which implementation of the method is called. To invoke a virtual method statically, the variable might have to be cast to a type that uses the desired version of the method. |

Overloading

Each type member has a unique signature. Method signatures consist of the method name and a parameter list (the order and types of the method's arguments). Multiple methods with the same name can be defined within a type as long as their signatures differ. When two or more methods with the same name are defined, the method is said to be overloaded. For example, in [System.Char](#), the [IsDigit](#) method is overloaded. One method takes a [Char](#). The other method takes a [String](#) and an [Int32](#).

 **Note**

The return type is not considered part of a method's signature. That is, methods cannot be overloaded if they differ only by return type.

Inherit, override, and hide members

A derived type inherits all members of its base type; that is, these members are defined on, and available to, the derived type. The behavior or qualities of inherited members can be modified in two ways:

- A derived type can hide an inherited member by defining a new member with the same signature. This might be done to make a previously public member private or to define new behavior for an inherited method that is marked as `final`.
- A derived type can override an inherited virtual method. The overriding method provides a new definition of the method that will be invoked based on the type of the value at run time rather than the type of the variable known at compile time. A method can override a virtual method only if the virtual method is not marked as `final` and the new method is at least as accessible as the virtual method.

See also

- [.NET API Browser](#)
- [Common Language Runtime](#)
- [Type Conversion in .NET](#)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

.NET is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

Language independence and language-independent components

Article • 12/21/2022

.NET is language independent. This means that, as a developer, you can develop in one of the many languages that target .NET implementations, such as C#, F#, and Visual Basic. You can access the types and members of class libraries developed for .NET implementations without having to know the language in which they were originally written and without having to follow any of the original language's conventions. If you're a component developer, your component can be accessed by any .NET app, regardless of its language.

ⓘ Note

This first part of this article discusses creating language-independent components, that is, components that can be consumed by apps that are written in any language. You can also create a single component or app from source code written in multiple languages; see [Cross-Language Interoperability](#) in the second part of this article.

To fully interact with other objects written in any language, objects must expose to callers only those features that are common to all languages. This common set of features is defined by the *Common Language Specification* (CLS), which is a set of rules that apply to generated assemblies. The Common Language Specification is defined in Partition I, Clauses 7 through 11 of the [ECMA-335 Standard: Common Language Infrastructure](#).

If your component conforms to the Common Language Specification, it is guaranteed to be CLS-compliant and can be accessed from code in assemblies written in any programming language that supports the CLS. You can determine whether your component conforms to the Common Language Specification at compile time by applying the [CLSCompliantAttribute](#) attribute to your source code. For more information, see [The CLSCompliantAttribute attribute](#).

CLS compliance rules

This section discusses the rules for creating a CLS-compliant component. For a complete list of rules, see Partition I, Clause 11 of the [ECMA-335 Standard: Common Language Infrastructure](#).

Note

The Common Language Specification discusses each rule for CLS compliance as it applies to consumers (developers who are programmatically accessing a component that is CLS-compliant), frameworks (developers who are using a language compiler to create CLS-compliant libraries), and extenders (developers who are creating a tool such as a language compiler or a code parser that creates CLS-compliant components). This article focuses on the rules as they apply to frameworks. Note, though, that some of the rules that apply to extenders may also apply to assemblies that are created using `Reflection.Emit`.

To design a component that's language independent, you only need to apply the rules for CLS compliance to your component's public interface. Your private implementation does not have to conform to the specification.

Important

The rules for CLS compliance apply only to a component's public interface, not to its private implementation.

For example, unsigned integers other than `Byte` are not CLS-compliant. Because the `Person` class in the following example exposes an `Age` property of type `UInt16`, the following code displays a compiler warning.

C#

```
using System;

[assembly: CLSCompliant(true)]

public class Person
{
    private UInt16 personAge = 0;

    public UInt16 Age
    { get { return personAge; } }

    // The attempt to compile the example displays the following compiler
    // warning:
    //   Public1.cs(10,18): warning CS3003: Type of 'Person.Age' is not CLS-
    //   compliant
```

You can make the `Person` class CLS-compliant by changing the type of the `Age` property from `UInt16` to `Int16`, which is a CLS-compliant, 16-bit signed integer. You don't have to

change the type of the private `personAge` field.

C#

```
using System;

[assembly: CLSCompliant(true)]

public class Person
{
    private Int16 personAge = 0;

    public Int16 Age
    { get { return personAge; } }
}
```

A library's public interface consists of the following:

- Definitions of public classes.
- Definitions of the public members of public classes, and definitions of members accessible to derived classes (that is, protected members).
- Parameters and return types of public methods of public classes, and parameters and return types of methods accessible to derived classes.

The rules for CLS compliance are listed in the following table. The text of the rules is taken verbatim from the [ECMA-335 Standard: Common Language Infrastructure](#), which is Copyright 2012 by Ecma International. More detailed information about these rules is found in the following sections.

[+] Expand table

| Category | See | Rule | Rule Number |
|---------------|--------------------------------------|--|-------------|
| Accessibility | Member accessibility | Accessibility shall not be changed when overriding inherited methods, except when overriding a method inherited from a different assembly with accessibility <code>family-or-assembly</code> . In this case, the override shall have accessibility <code>family</code> . | 10 |
| Accessibility | Member accessibility | The visibility and accessibility of types and members shall be such that types in the signature of any member shall be visible and accessible whenever the member itself is visible and accessible. For example, a public method that is visible outside its assembly shall not have an argument whose type is visible | 12 |

| Category | See | Rule | Rule Number |
|--------------|------------------------------|--|-------------|
| | | only within the assembly. The visibility and accessibility of types composing an instantiated generic type used in the signature of any member shall be visible and accessible whenever the member itself is visible and accessible. For example, an instantiated generic type present in the signature of a member that is visible outside its assembly shall not have a generic argument whose type is visible only within the assembly. | |
| Arrays | Arrays | Arrays shall have elements with a CLS-compliant type, and all dimensions of the array shall have lower bounds of zero. Only the fact that an item is an array and the element type of the array shall be required to distinguish between overloads. When overloading is based on two or more array types, the element types shall be named types. | 16 |
| Attributes | Attributes | Attributes shall be of type System.Attribute , or a type inheriting from it. | 41 |
| Attributes | Attributes | The CLS only allows a subset of the encodings of custom attributes. The only types that shall appear in these encodings are (see Partition IV): System.Type , System.String , System.Char , System.Boolean , System.Byte , System.Int16 , System.Int32 , System.Int64 , System.Single , System.Double , and any enumeration type based on a CLS-compliant base integer type. | 34 |
| Attributes | Attributes | The CLS does not allow publicly visible required modifiers (<code>modreq</code> , see Partition II), but does allow optional modifiers (<code>modopt</code> , see Partition II) it does not understand. | 35 |
| Constructors | Constructors | An object constructor shall call some instance constructor of its base class before any access occurs to inherited instance data. (This does not apply to value types, which need not have constructors.) | 21 |
| Constructors | Constructors | An object constructor shall not be called except as part of the creation of an object, and an object shall not be initialized twice. | 22 |
| Enumerations | Enumerations | The underlying type of an enum shall be a built-in CLS integer type, the name of the field shall be | 7 |

| Category | See | Rule | Rule Number |
|--------------|--------------------------------------|--|-------------|
| | | | |
| Enumerations | Enumerations | There are two distinct kinds of enums, indicated by the presence or absence of the System.FlagsAttribute (see Partition IV Library) custom attribute. One represents named integer values; the other represents named bit flags that can be combined to generate an unnamed value. The value of an <code>enum</code> is not limited to the specified values. | 8 |
| Enumerations | Enumerations | Literal static fields of an enum shall have the type of the enum itself. | 9 |
| Events | Events | The methods that implement an event shall be marked <code>SpecialName</code> in the metadata. | 29 |
| Events | Events | The accessibility of an event and of its accessors shall be identical. | 30 |
| Events | Events | The <code>add</code> and <code>remove</code> methods for an event shall both either be present or absent. | 31 |
| Events | Events | The <code>add</code> and <code>remove</code> methods for an event shall each take one parameter whose type defines the type of the event and that shall be derived from System.Delegate . | 32 |
| Events | Events | Events shall adhere to a specific naming pattern. The <code>SpecialName</code> attribute referred to in CLS rule 29 shall be ignored in appropriate name comparisons and shall adhere to identifier rules. | 33 |
| Exceptions | Exceptions | Objects that are thrown shall be of type System.Exception or a type inheriting from it. Nonetheless, CLS-compliant methods are not required to block the propagation of other types of exceptions. | 40 |
| General | CLS compliance rules | CLS rules apply only to those parts of a type that are accessible or visible outside of the defining assembly. | 1 |
| General | CLS compliance rules | Members of non-CLS compliant types shall not be marked CLS-compliant. | 2 |

| Category | See | Rule | Rule Number |
|------------|---|--|-------------|
| Generics | Generic types and members | Nested types shall have at least as many generic parameters as the enclosing type. Generic parameters in a nested type correspond by position to the generic parameters in its enclosing type. | 42 |
| Generics | Generic types and members | The name of a generic type shall encode the number of type parameters declared on the non-nested type, or newly introduced to the type if nested, according to the rules defined above. | 43 |
| Generics | Generic types and members | A generic type shall redeclare sufficient constraints to guarantee that any constraints on the base type, or interfaces would be satisfied by the generic type constraints. | 44 |
| Generics | Generic types and members | Types used as constraints on generic parameters shall themselves be CLS-compliant. | 45 |
| Generics | Generic types and members | The visibility and accessibility of members (including nested types) in an instantiated generic type shall be considered to be scoped to the specific instantiation rather than the generic type declaration as a whole. Assuming this, the visibility and accessibility rules of CLS rule 12 still apply. | 46 |
| Generics | Generic types and members | For each abstract or virtual generic method, there shall be a default concrete (nonabstract) implementation | 47 |
| Interfaces | Interfaces | CLS-compliant interfaces shall not require the definition of non-CLS compliant methods in order to implement them. | 18 |
| Interfaces | Interfaces | CLS-compliant interfaces shall not define static methods, nor shall they define fields. | 19 |
| Members | Type members in general | Global static fields and methods are not CLS-compliant. | 36 |
| Members | -- | The value of a literal static is specified by using field initialization metadata. A CLS-compliant literal must have a value specified in field initialization metadata that is of exactly the same type as the literal (or of the underlying type, if that literal is an <code>enum</code>). | 13 |
| Members | Type members in | The vararg constraint is not part of the CLS, and the only calling convention supported by the CLS is the | 15 |

| Category | See | Rule | Rule Number |
|--------------------|------------------------------------|---|-------------|
| | general | standard managed calling convention. | |
| Naming conventions | Naming conventions | Assemblies shall follow Annex 7 of Technical Report 15 of the Unicode Standard 3.0 governing the set of characters permitted to start and be included in identifiers, available online at Unicode Normalization Forms . Identifiers shall be in the canonical format defined by Unicode Normalization Form C. For CLS purposes, two identifiers are the same if their lowercase mappings (as specified by the Unicode locale-insensitive, one-to-one lowercase mappings) are the same. That is, for two identifiers to be considered different under the CLS they shall differ in more than simply their case. However, in order to override an inherited definition the CLI requires the precise encoding of the original declaration be used. | 4 |
| Overloading | Naming conventions | All names introduced in a CLS-compliant scope shall be distinct independent of kind, except where the names are identical and resolved via overloading. That is, while the CTS allows a single type to use the same name for a method and a field, the CLS does not. | 5 |
| Overloading | Naming conventions | Fields and nested types shall be distinct by identifier comparison alone, even though the CTS allows distinct signatures to be distinguished. Methods, properties, and events that have the same name (by identifier comparison) shall differ by more than just the return type, except as specified in CLS Rule 39 | 6 |
| Overloading | Overloads | Only properties and methods can be overloaded. | 37 |
| Overloading | Overloads | Properties and methods can be overloaded based only on the number and types of their parameters, except the conversion operators named <code>op_Implicit</code> and <code>op_Explicit</code> , which can also be overloaded based on their return type. | 38 |
| Overloading | -- | If two or more CLS-compliant methods declared in a type have the same name and, for a specific set of type instantiations, they have the same parameter and return types, then all these methods shall be semantically equivalent at those type instantiations. | 48 |
| Properties | Properties | The methods that implement the getter and setter methods of a property shall be marked <code>SpecialName</code> | 24 |

| Category | See | Rule | Rule Number | |
|-----------------|--|---|-------------|--|
| | | in the metadata. | | |
| Properties | Properties | A property's accessors shall all be static, all be virtual, or all be instance. | 26 | |
| Properties | Properties | The type of a property shall be the return type of the getter and the type of the last argument of the setter. The types of the parameters of the property shall be the types of the parameters to the getter and the types of all but the final parameter of the setter. All of these types shall be CLS-compliant, and shall not be managed pointers (that is, shall not be passed by reference). | 27 | |
| Properties | Properties | Properties shall adhere to a specific naming pattern. The <code>SpecialName</code> attribute referred to in CLS rule 24 shall be ignored in appropriate name comparisons and shall adhere to identifier rules. A property shall have a getter method, a setter method, or both. | 28 | |
| Type conversion | Type conversion | If either <code>op_Implicit</code> or <code>op_Explicit</code> is provided, an alternate means of providing the coercion shall be provided. | 39 | |
| Types | Types and type member signatures | Boxed value types are not CLS-compliant. | 3 | |
| Types | Types and type member signatures | All types appearing in a signature shall be CLS-compliant. All types composing an instantiated generic type shall be CLS-compliant. | 11 | |
| Types | Types and type member signatures | Typed references are not CLS-compliant. | 14 | |
| Types | Types and type member signatures | Unmanaged pointer types are not CLS-compliant. | 17 | |
| Types | Types and type member signatures | CLS-compliant classes, value types, and interfaces shall not require the implementation of non-CLS-compliant members | 20 | |
| Types | Types and type member signatures | <code>System.Object</code> is CLS-compliant. Any other CLS-compliant class shall inherit from a CLS-compliant class. | 23 | |

Index to subsections:

- [Types and type member signatures](#)
- [Naming conventions](#)
- [Type conversion](#)
- [Arrays](#)
- [Interfaces](#)
- [Enumerations](#)
- [Type members in general](#)
- [Member accessibility](#)
- [Generic types and members](#)
- [Constructors](#)
- [Properties](#)
- [Events](#)
- [Overloads](#)
- [Exceptions](#)
- [Attributes](#)

Types and type-member signatures

The [System.Object](#) type is CLS-compliant and is the base type of all object types in the .NET type system. Inheritance in .NET is either implicit (for example, the [String](#) class implicitly inherits from the `Object` class) or explicit (for example, the [CultureNotFoundException](#) class explicitly inherits from the [ArgumentException](#) class, which explicitly inherits from the [Exception](#) class). For a derived type to be CLS compliant, its base type must also be CLS-compliant.

The following example shows a derived type whose base type is not CLS-compliant. It defines a base `Counter` class that uses an unsigned 32-bit integer as a counter. Because the class provides counter functionality by wrapping an unsigned integer, the class is marked as non-CLS-compliant. As a result, a derived class, `NonZeroCounter`, is also not CLS-compliant.

C#

```
using System;

[assembly: CLSCompliant(true)]

[CLSCompliant(false)]
public class Counter
{
    UInt32 ctr;
```

```

public Counter()
{
    ctr = 0;
}

protected Counter(UInt32 ctr)
{
    this.ctr = ctr;
}

public override string ToString()
{
    return String.Format("{0}). ", ctr);
}

public UInt32 Value
{
    get { return ctr; }
}

public void Increment()
{
    ctr += (uint) 1;
}

public class NonZeroCounter : Counter
{
    public NonZeroCounter(int startIndex) : this((uint) startIndex)
    {
    }

    private NonZeroCounter(UInt32 startIndex) : base(startIndex)
    {
    }
}

// Compilation produces a compiler warning like the following:
// Type3.cs(37,14): warning CS3009: 'NonZeroCounter': base type 'Counter'
// is not
//           CLS-compliant
// Type3.cs(7,14): (Location of symbol related to previous warning)

```

All types that appear in member signatures, including a method's return type or a property type, must be CLS-compliant. In addition, for generic types:

- All types that compose an instantiated generic type must be CLS-compliant.
- All types used as constraints on generic parameters must be CLS-compliant.

The .NET [common type system](#) includes many built-in types that are supported directly by the common language runtime and are specially encoded in an assembly's metadata. Of these intrinsic types, the types listed in the following table are CLS-compliant.

[Expand table](#)

| CLS-compliant type | Description |
|--------------------|---|
| Byte | 8-bit unsigned integer |
| Int16 | 16-bit signed integer |
| Int32 | 32-bit signed integer |
| Int64 | 64-bit signed integer |
| Half | Half-precision floating-point value |
| Single | Single-precision floating-point value |
| Double | Double-precision floating-point value |
| Boolean | true or false value type |
| Char | UTF-16 encoded code unit |
| Decimal | Non-floating-point decimal number |
| IntPtr | Pointer or handle of a platform-defined size |
| String | Collection of zero, one, or more Char objects |

The intrinsic types listed in the following table are not CLS-Compliant.

[Expand table](#)

| Non-compliant type | Description | CLS-compliant alternative |
|--------------------|--------------------------------|---|
| SByte | 8-bit signed integer data type | Int16 |
| UInt16 | 16-bit unsigned integer | Int32 |
| UInt32 | 32-bit unsigned integer | Int64 |
| UInt64 | 64-bit unsigned integer | Int64 (may overflow), BigInteger, or Double |
| IntPtr | Unsigned pointer or handle | IntPtr |

The .NET Class Library or any other class library may include other types that aren't CLS-compliant, for example:

- Boxed value types. The following C# example creates a class that has a public property of type `int*` named `Value`. Because an `int*` is a boxed value type, the compiler flags it as non-CLS-compliant.

```
C#
using System;

[assembly: CLSCompliant(true)]

public unsafe class TestClass
{
    private int* val;

    public TestClass(int number)
    {
        val = (int*) number;
    }

    public int* Value {
        get { return val; }
    }
}

// The compiler generates the following output when compiling this
// example:
//      warning CS3003: Type of 'TestClass.Value' is not CLS-
//      compliant
```

- Typed references, which are special constructs that contain a reference to an object and a reference to a type. Typed references are represented in .NET by the [TypedReference](#) class.

If a type is not CLS-compliant, you should apply the [CLSCompliantAttribute](#) attribute with an `isCompliant` value of `false` to it. For more information, see [The CLSCompliantAttribute attribute](#) section.

The following example illustrates the problem of CLS compliance in a method signature and in generic type instantiation. It defines an `InvoiceItem` class with a property of type `UInt32`, a property of type `Nullable<UInt32>`, and a constructor with parameters of type `UInt32` and `Nullable<UInt32>`. You get four compiler warnings when you try to compile this example.

```
C#
using System;

[assembly: CLSCompliant(true)]
```

```

public class InvoiceItem
{
    private uint invId = 0;
    private uint itemId = 0;
    private Nullable<uint> qty;

    public InvoiceItem(uint sku, Nullable<uint> quantity)
    {
        itemId = sku;
        qty = quantity;
    }

    public Nullable<uint> Quantity
    {
        get { return qty; }
        set { qty = value; }
    }

    public uint InvoiceId
    {
        get { return invId; }
        set { invId = value; }
    }
}

// The attempt to compile the example displays the following output:
//   Type1.cs(13,23): warning CS3001: Argument type 'uint' is not CLS-
//   compliant
//   Type1.cs(13,33): warning CS3001: Argument type 'uint?' is not CLS-
//   compliant
//   Type1.cs(19,26): warning CS3003: Type of 'InvoiceItem.Quantity' is not
//   CLS-compliant
//   Type1.cs(25,16): warning CS3003: Type of 'InvoiceItem.InvoiceId' is
//   not CLS-compliant

```

To eliminate the compiler warnings, replace the non-CLS-compliant types in the `InvoiceItem` public interface with compliant types:

C#

```

using System;

[assembly: CLSCompliant(true)]

public class InvoiceItem
{
    private uint invId = 0;
    private uint itemId = 0;
    private Nullable<int> qty;

    public InvoiceItem(int sku, Nullable<int> quantity)
    {
        if (sku <= 0)
            throw new ArgumentOutOfRangeException("The item number is zero or

```

```

    negative.");
    itemId = (uint) sku;

    qty = quantity;
}

public Nullable<int> Quantity
{
    get { return qty; }
    set { qty = value; }
}

public int InvoiceId
{
    get { return (int) invId; }
    set {
        if (value <= 0)
            throw new ArgumentOutOfRangeException("The invoice number is
zero or negative.");
        invId = (uint) value; }
}
}

```

In addition to the specific types listed, some categories of types are not CLS compliant. These include unmanaged pointer types and function pointer types. The following example generates a compiler warning because it uses a pointer to an integer to create an array of integers.

C#

```

using System;

[assembly: CLSCompliant(true)]

public class ArrayHelper
{
    unsafe public static Array CreateInstance(Type type, int* ptr, int items)
    {
        Array arr = Array.CreateInstance(type, items);
        int* addr = ptr;
        for (int ctr = 0; ctr < items; ctr++) {
            int value = *addr;
            arr.SetValue(value, ctr);
            addr++;
        }
        return arr;
    }
}

// The attempt to compile this example displays the following output:
//     UnmanagedType1.cs(8,57): warning CS3001: Argument type 'int*' is not
CLS-compliant

```

For CLS-compliant abstract classes (that is, classes marked as `abstract` in C# or as `MustInherit` in Visual Basic), all members of the class must also be CLS-compliant.

Naming conventions

Because some programming languages are case-insensitive, identifiers (such as the names of namespaces, types, and members) must differ by more than case. Two identifiers are considered equivalent if their lowercase mappings are the same. The following C# example defines two public classes, `Person` and `person`. Because they differ only by case, the C# compiler flags them as not CLS-compliant.

C#

```
using System;

[assembly: CLSCompliant(true)]

public class Person : person
{
}

public class person
{
}

// Compilation produces a compiler warning like the following:
//   Naming1.cs(11,14): warning CS3005: Identifier 'person' differing
//                     only in case is not CLS-compliant
//   Naming1.cs(6,14): (Location of symbol related to previous warning)
```

Programming language identifiers, such as the names of namespaces, types, and members, must conform to the [Unicode Standard](#). This means that:

- The first character of an identifier can be any Unicode uppercase letter, lowercase letter, title case letter, modifier letter, other letter, or letter number. For information on Unicode character categories, see the [System.Globalization.UnicodeCategory](#) enumeration.
- Subsequent characters can be from any of the categories as the first character, and can also include non-spacing marks, spacing combining marks, decimal numbers, connector punctuation, and formatting codes.

Before you compare identifiers, you should filter out formatting codes and convert the identifiers to Unicode Normalization Form C, because a single character can be represented by multiple UTF-16-encoded code units. Character sequences that produce the same code units in Unicode Normalization Form C are not CLS-compliant. The

following example defines a property named Å, which consists of the character ANGSTROM SIGN (U+212B), and a second property named Å, which consists of the character LATIN CAPITAL LETTER A WITH RING ABOVE (U+00C5). Both the C# and Visual Basic compilers flag the source code as non-CLS-compliant.

C#

```
public class Size
{
    private double a1;
    private double a2;

    public double Å
    {
        get { return a1; }
        set { a1 = value; }
    }

    public double å
    {
        get { return a2; }
        set { a2 = value; }
    }
}

// Compilation produces a compiler warning like the following:
//     Naming2a.cs(16,18): warning CS3005: Identifier 'Size.Å' differing only
in case is not
//         CLS-compliant
//     Naming2a.cs(10,18): (Location of symbol related to previous warning)
//     Naming2a.cs(18,8): warning CS3005: Identifier 'Size.Å.get' differing
only in case is not
//         CLS-compliant
//     Naming2a.cs(12,8): (Location of symbol related to previous warning)
//     Naming2a.cs(19,8): warning CS3005: Identifier 'Size.Å.set' differing
only in case is not
//         CLS-compliant
//     Naming2a.cs(13,8): (Location of symbol related to previous warning)
```

Member names within a particular scope (such as the namespaces within an assembly, the types within a namespace, or the members within a type) must be unique except for names that are resolved through overloading. This requirement is more stringent than that of the common type system, which allows multiple members within a scope to have identical names as long as they are different kinds of members (for example, one is a method and one is a field). In particular, for type members:

- Fields and nested types are distinguished by name alone.
- Methods, properties, and events that have the same name must differ by more than just return type.

The following example illustrates the requirement that member names must be unique within their scope. It defines a class named `Converter` that includes four members named `Conversion`. Three are methods, and one is a property. The method that includes an `Int64` parameter is uniquely named, but the two methods with an `Int32` parameter are not, because return value is not considered a part of a member's signature. The `Conversion` property also violates this requirement, because properties cannot have the same name as overloaded methods.

C#

```
using System;

[assembly: CLSCompliant(true)]

public class Converter
{
    public double Conversion(int number)
    {
        return (double) number;
    }

    public float Conversion(int number)
    {
        return (float) number;
    }

    public double Conversion(long number)
    {
        return (double) number;
    }

    public bool Conversion
    {
        get { return true; }
    }
}

// Compilation produces a compiler error like the following:
// Naming3.cs(13,17): error CS0111: Type 'Converter' already defines a
// member called
//         'Conversion' with the same parameter types
// Naming3.cs(8,18): (Location of symbol related to previous error)
// Naming3.cs(23,16): error CS0102: The type 'Converter' already contains
// a definition for
//         'Conversion'
// Naming3.cs(8,18): (Location of symbol related to previous error)
```

Individual languages include unique keywords, so languages that target the common language runtime must also provide some mechanism for referencing identifiers (such as type names) that coincide with keywords. For example, `case` is a keyword in both C# and VB.

and Visual Basic. However, the following Visual Basic example is able to disambiguate a class named `case` from the `case` keyword by using opening and closing braces. Otherwise, the example would produce the error message, "Keyword is not valid as an identifier," and fail to compile.

VB

```
Public Class [case]
    Private _id As Guid
    Private name As String

    Public Sub New(name As String)
        _id = Guid.NewGuid()
        Me.name = name
    End Sub

    Public ReadOnly Property ClientName As String
        Get
            Return name
        End Get
    End Property
End Class
```

The following C# example is able to instantiate the `case` class by using the `@` symbol to disambiguate the identifier from the language keyword. Without it, the C# compiler would display two error messages, "Type expected" and "Invalid expression term 'case'."

C#

```
using System;

public class Example
{
    public static void Main()
    {
        @case c = new @case("John");
        Console.WriteLine(c.ClientName);
    }
}
```

Type conversion

The Common Language Specification defines two conversion operators:

- `op_Implicit`, which is used for widening conversions that do not result in loss of data or precision. For example, the `Decimal` structure includes an overloaded

`op_Implicit` operator to convert values of integral types and `Char` values to `Decimal` values.

- `op_Explicit`, which is used for narrowing conversions that can result in loss of magnitude (a value is converted to a value that has a smaller range) or precision. For example, the `Decimal` structure includes an overloaded `op_Explicit` operator to convert `Double` and `Single` values to `Decimal` and to convert `Decimal` values to integral values, `Double`, `Single`, and `Char`.

However, not all languages support operator overloading or the definition of custom operators. If you choose to implement these conversion operators, you should also provide an alternate way to perform the conversion. We recommend that you provide `FromXxx` and `ToXxx` methods.

The following example defines CLS-compliant implicit and explicit conversions. It creates a `UDouble` class that represents an unsigned, double-precision, floating-point number. It provides for implicit conversions from `UDouble` to `Double` and for explicit conversions from `UDouble` to `Single`, `Double` to `UDouble`, and `Single` to `UDouble`. It also defines a `ToDouble` method as an alternative to the implicit conversion operator and the `ToSingle`, `FromDouble`, and `FromSingle` methods as alternatives to the explicit conversion operators.

C#

```
using System;

public struct UDouble
{
    private double number;

    public UDouble(double value)
    {
        if (value < 0)
            throw new InvalidCastException("A negative value cannot be
converted to a UDouble.");

        number = value;
    }

    public UDouble(float value)
    {
        if (value < 0)
            throw new InvalidCastException("A negative value cannot be
converted to a UDouble.");

        number = value;
    }
}
```

```
public static readonly UDouble MinValue = (UDouble) 0.0;
public static readonly UDouble MaxValue = (UDouble) Double.MaxValue;

public static explicit operator Double(UDouble value)
{
    return value.number;
}

public static implicit operator Single(UDouble value)
{
    if (value.number > (double) Single.MaxValue)
        throw new InvalidCastException("A UDouble value is out of range of
the Single type.");

    return (float) value.number;
}

public static explicit operator UDouble(double value)
{
    if (value < 0)
        throw new InvalidCastException("A negative value cannot be
converted to a UDouble.");

    return new UDouble(value);
}

public static implicit operator UDouble(float value)
{
    if (value < 0)
        throw new InvalidCastException("A negative value cannot be
converted to a UDouble.");

    return new UDouble(value);
}

public static Double ToDouble(UDouble value)
{
    return (Double) value;
}

public static float ToSingle(UDouble value)
{
    return (float) value;
}

public static UDouble FromDouble(double value)
{
    return new UDouble(value);
}

public static UDouble FromSingle(float value)
{
    return new UDouble(value);
}
```

Arrays

CLS-compliant arrays conform to the following rules:

- All dimensions of an array must have a lower bound of zero. The following example creates a non-CLS-compliant array with a lower bound of one. Despite the presence of the [CLSCompliantAttribute](#) attribute, the compiler does not detect that the array returned by the `Numbers.GetTenPrimes` method is not CLS-compliant.

C#

```
[assembly: CLSCompliant(true)]  
  
public class Numbers  
{  
    public static Array GetTenPrimes()  
    {  
        Array arr = Array.CreateInstance(typeof(Int32), new int[] {10},  
new int[] {1});  
        arr.SetValue(1, 1);  
        arr.SetValue(2, 2);  
        arr.SetValue(3, 3);  
        arr.SetValue(5, 4);  
        arr.SetValue(7, 5);  
        arr.SetValue(11, 6);  
        arr.SetValue(13, 7);  
        arr.SetValue(17, 8);  
        arr.SetValue(19, 9);  
        arr.SetValue(23, 10);  
  
        return arr;  
    }  
}
```

- All array elements must consist of CLS-compliant types. The following example defines two methods that return non-CLS-compliant arrays. The first returns an array of [UInt32](#) values. The second returns an [Object](#) array that includes [Int32](#) and [UInt32](#) values. Although the compiler identifies the first array as non-compliant because of its [UInt32](#) type, it fails to recognize that the second array includes non-CLS-compliant elements.

C#

```
using System;  
  
[assembly: CLSCompliant(true)]
```

```

public class Numbers
{
    public static UInt32[] GetTenPrimes()
    {
        uint[] arr = { 1u, 2u, 3u, 5u, 7u, 11u, 13u, 17u, 19u };
        return arr;
    }

    public static Object[] GetFivePrimes()
    {
        Object[] arr = { 1, 2, 3, 5u, 7u };
        return arr;
    }
}

// Compilation produces a compiler warning like the following:
//     Array2.cs(8,27): warning CS3002: Return type of
'Numbers.GetTenPrimes()' is not
//                 CLS-compliant

```

- Overload resolution for methods that have array parameters is based on the fact that they are arrays and on their element type. For this reason, the following definition of an overloaded `GetSquares` method is CLS-compliant.

C#

```

using System;
using System.Numerics;

[assembly: CLSCompliant(true)]

public class Numbers
{
    public static byte[] GetSquares(byte[] numbers)
    {
        byte[] numbersOut = new byte[numbers.Length];
        for (int ctr = 0; ctr < numbers.Length; ctr++) {
            int square = ((int) numbers[ctr]) * ((int) numbers[ctr]);
            if (square <= Byte.MaxValue)
                numbersOut[ctr] = (byte) square;
            // If there's an overflow, assign MaxValue to the
            corresponding
            // element.
            else
                numbersOut[ctr] = Byte.MaxValue;
        }
        return numbersOut;
    }

    public static BigInteger[] GetSquares(BigInteger[] numbers)
    {
        BigInteger[] numbersOut = new BigInteger[numbers.Length];

```

```

        for (int ctr = 0; ctr < numbers.Length; ctr++)
            numbersOut[ctr] = numbers[ctr] * numbers[ctr];

        return numbersOut;
    }
}

```

Interfaces

CLS-compliant interfaces can define properties, events, and virtual methods (methods with no implementation). A CLS-compliant interface cannot have any of the following:

- Static methods or static fields. Both the C# and Visual Basic compilers generate compiler errors if you define a static member in an interface.
- Fields. Both the C# and Visual Basic compilers generate compiler errors if you define a field in an interface.
- Methods that are not CLS-compliant. For example, the following interface definition includes a method, `INumber.GetUnsigned`, that is marked as non-CLS-compliant. This example generates a compiler warning.

C#

```

using System;

[assembly:CLSCCompliant(true)]

public interface INumber
{
    int Length();
    [CLSCCompliant(false)] ulong GetUnsigned();
}

// Attempting to compile the example displays output like the
// following:
//  Interface2.cs(8,32): warning CS3010: 'INumber.GetUnsigned()':
//  CLS-compliant interfaces
//                      must have only CLS-compliant members

```

Because of this rule, CLS-compliant types are not required to implement non-CLS-compliant members. If a CLS-compliant framework does expose a class that implements a non-CLS compliant interface, it should also provide concrete implementations of all non-CLS-compliant members.

CLS-compliant language compilers must also allow a class to provide separate implementations of members that have the same name and signature in multiple

interfaces. Both C# and Visual Basic support [explicit interface implementations](#) to provide different implementations of identically named methods. Visual Basic also supports the `Implements` keyword, which enables you to explicitly designate which interface and member a particular member implements. The following example illustrates this scenario by defining a `Temperature` class that implements the `ICelsius` and `IFahrenheit` interfaces as explicit interface implementations.

C#

```
using System;

[assembly: CLSCompliant(true)]

public interface IFahrenheit
{
    decimal GetTemperature();
}

public interface ICelsius
{
    decimal GetTemperature();
}

public class Temperature : ICelsius, IFahrenheit
{
    private decimal _value;

    public Temperature(decimal value)
    {
        // We assume that this is the Celsius value.
        _value = value;
    }

    decimal IFahrenheit.GetTemperature()
    {
        return _value * 9 / 5 + 32;
    }

    decimal ICelsius.GetTemperature()
    {
        return _value;
    }
}

public class Example
{
    public static void Main()
    {
        Temperature temp = new Temperature(100.0m);
        ICelsius cTemp = temp;
        IFahrenheit fTemp = temp;
        Console.WriteLine("Temperature in Celsius: {0} degrees",
            cTemp.GetTemperature());
```

```

        Console.WriteLine("Temperature in Fahrenheit: {0} degrees",
                           fTemp.GetTemperature());
    }
}

// The example displays the following output:
//      Temperature in Celsius: 100.0 degrees
//      Temperature in Fahrenheit: 212.0 degrees

```

Enumerations

CLS-compliant enumerations must follow these rules:

- The underlying type of the enumeration must be an intrinsic CLS-compliant integer ([Byte](#), [Int16](#), [Int32](#), or [Int64](#)). For example, the following code tries to define an enumeration whose underlying type is [UInt32](#) and generates a compiler warning.

C#

```

using System;

[assembly: CLSCompliant(true)]

public enum Size : uint {
    Unspecified = 0,
    XSmall = 1,
    Small = 2,
    Medium = 3,
    Large = 4,
    XLarge = 5
};

public class Clothing
{
    public string Name;
    public string Type;
    public string Size;
}

// The attempt to compile the example displays a compiler warning like
// the following:
//  Enum3.cs(6,13): warning CS3009: 'Size': base type 'uint' is not
//  CLS-compliant

```

- An enumeration type must have a single instance field named `value_` that is marked with the [FieldAttributes.RTSpecialName](#) attribute. This enables you to reference the field value implicitly.
- An enumeration includes literal static fields whose types match the type of the enumeration itself. For example, if you define a `State` enumeration with values of

`State.On` and `State.Off`, `State.On` and `State.Off` are both literal static fields whose type is `State`.

- There are two kinds of enumerations:
 - An enumeration that represents a set of mutually exclusive, named integer values. This type of enumeration is indicated by the absence of the [System.FlagsAttribute](#) custom attribute.
 - An enumeration that represents a set of bit flags that can combine to generate an unnamed value. This type of enumeration is indicated by the presence of the [System.FlagsAttribute](#) custom attribute.

For more information, see the documentation for the [Enum](#) structure.

- The value of an enumeration is not limited to the range of its specified values. In other words, the range of values in an enumeration is the range of its underlying value. You can use the [Enum.IsDefined](#) method to determine whether a specified value is a member of an enumeration.

Type members in general

The Common Language Specification requires all fields and methods to be accessed as members of a particular class. Therefore, global static fields and methods (that is, static fields or methods that are defined apart from a type) are not CLS-compliant. If you try to include a global field or method in your source code, both the C# and Visual Basic compilers generate a compiler error.

The Common Language Specification supports only the standard managed calling convention. It doesn't support unmanaged calling conventions and methods with variable argument lists marked with the `varargs` keyword. For variable argument lists that are compatible with the standard managed calling convention, use the [ParamArrayAttribute](#) attribute or the individual language's implementation, such as the `params` keyword in C# and the `ParamArray` keyword in Visual Basic.

Member accessibility

Overriding an inherited member cannot change the accessibility of that member. For example, a public method in a base class cannot be overridden by a private method in a derived class. There is one exception: a `protected internal` (in C#) or `Protected Friend` (in Visual Basic) member in one assembly that is overridden by a type in a different assembly. In that case, the accessibility of the override is `Protected`.

The following example illustrates the error that is generated when the [CLSCompliantAttribute](#) attribute is set to `true`, and `Human`, which is a class derived from `Animal`, tries to change the accessibility of the `Species` property from public to private. The example compiles successfully if its accessibility is changed to public.

C#

```
using System;

[assembly: CLSCompliant(true)]

public class Animal
{
    private string _species;

    public Animal(string species)
    {
        _species = species;
    }

    public virtual string Species
    {
        get { return _species; }
    }

    public override string ToString()
    {
        return _species;
    }
}

public class Human : Animal
{
    private string _name;

    public Human(string name) : base("Homo Sapiens")
    {
        _name = name;
    }

    public string Name
    {
        get { return _name; }
    }

    private override string Species
    {
        get { return base.Species; }
    }

    public override string ToString()
    {
        return _name;
    }
}
```

```

    }

}

public class Example
{
    public static void Main()
    {
        Human p = new Human("John");
        Console.WriteLine(p.Species);
        Console.WriteLine(p.ToString());
    }
}

// The example displays the following output:
//      error CS0621: 'Human.Species': virtual or abstract members cannot be
private

```

Types in the signature of a member must be accessible whenever that member is accessible. For example, this means that a public member cannot include a parameter whose type is private, protected, or internal. The following example illustrates the compiler error that results when a `StringWrapper` class constructor exposes an internal `StringOperationType` enumeration value that determines how a string value should be wrapped.

C#

```

using System;
using System.Text;

public class StringWrapper
{
    string internalString;
    StringBuilder internalSB = null;
    bool useSB = false;

    public StringWrapper(StringOperationType type)
    {
        if (type == StringOperationType.Normal) {
            useSB = false;
        }
        else {
            useSB = true;
            internalSB = new StringBuilder();
        }
    }

    // The remaining source code...
}

internal enum StringOperationType { Normal, Dynamic }
// The attempt to compile the example displays the following output:
//      error CS0051: Inconsistent accessibility: parameter type

```

```
//           'StringOperationType' is less accessible than method
//           'StringWrapper.StringWrapper(StringOperationType)'
```

Generic types and members

Nested types always have at least as many generic parameters as their enclosing type. These correspond by position to the generic parameters in the enclosing type. The generic type can also include new generic parameters.

The relationship between the generic type parameters of a containing type and its nested types may be hidden by the syntax of individual languages. In the following example, a generic type `Outer<T>` contains two nested classes, `Inner1A` and `Inner1B<U>`. The calls to the `ToString` method, which each class inherits from `Object.ToString()`, show that each nested class includes the type parameters of its containing class.

C#

```
using System;

[assembly:CLSCCompliant(true)]

public class Outer<T>
{
    T value;

    public Outer(T value)
    {
        this.value = value;
    }

    public class Inner1A : Outer<T>
    {
        public Inner1A(T value) : base(value)
        { }
    }

    public class Inner1B<U> : Outer<T>
    {
        U value2;

        public Inner1B(T value1, U value2) : base(value1)
        {
            this.value2 = value2;
        }
    }
}

public class Example
```

```

public static void Main()
{
    var inst1 = new Outer<String>("This");
    Console.WriteLine(inst1);

    var inst2 = new Outer<String>.Inner1A("Another");
    Console.WriteLine(inst2);

    var inst3 = new Outer<String>.Inner1B<int>("That", 2);
    Console.WriteLine(inst3);
}

// The example displays the following output:
//      Outer`1[System.String]
//      Outer`1+Inner1A[System.String]
//      Outer`1+Inner1B`1[System.String,System.Int32]

```

Generic type names are encoded in the form *name`n*, where *name* is the type name, ` is a character literal, and *n* is the number of parameters declared on the type, or, for nested generic types, the number of newly introduced type parameters. This encoding of generic type names is primarily of interest to developers who use reflection to access CLS-compliant generic types in a library.

If constraints are applied to a generic type, any types used as constraints must also be CLS-compliant. The following example defines a class named `BaseClass` that is not CLS-compliant and a generic class named `BaseCollection` whose type parameter must derive from `BaseClass`. But because `BaseClass` is not CLS-compliant, the compiler emits a warning.

```

C#

using System;

[assembly:CLSCCompliant(true)]

[CLSCCompliant(false)] public class BaseClass
{ }

public class BaseCollection<T> where T : BaseClass
{ }
// Attempting to compile the example displays the following output:
//     warning CS3024: Constraint type 'BaseClass' is not CLS-compliant

```

If a generic type is derived from a generic base type, it must redeclare any constraints so that it can guarantee that constraints on the base type are also satisfied. The following example defines a `Number<T>` that can represent any numeric type. It also defines a `FloatingPoint<T>` class that represents a floating-point value. However, the source code

fails to compile, because it does not apply the constraint on `Number<T>` (that `T` must be a value type) to `FloatingPoint<T>`.

C#

```
using System;

[assembly:CLSCCompliant(true)]

public class Number<T> where T : struct
{
    // use Double as the underlying type, since its range is a superset of
    // the ranges of all numeric types except BigInteger.
    protected double number;

    public Number(T value)
    {
        try {
            this.number = Convert.ToDouble(value);
        }
        catch (OverflowException e) {
            throw new ArgumentException("value is too large.", e);
        }
        catch (InvalidCastException e) {
            throw new ArgumentException("The value parameter is not numeric.", e);
        }
    }

    public T Add(T value)
    {
        return (T) Convert.ChangeType(number + Convert.ToDouble(value),
typeof(T));
    }

    public T Subtract(T value)
    {
        return (T) Convert.ChangeType(number - Convert.ToDouble(value),
typeof(T));
    }
}

public class FloatingPoint<T> : Number<T>
{
    public FloatingPoint(T number) : base(number)
    {
        if (typeof(float) == number.GetType() ||
            typeof(double) == number.GetType() ||
            typeof(decimal) == number.GetType())
            this.number = Convert.ToDouble(number);
        else
            throw new ArgumentException("The number parameter is not a
floating-point number.");
    }
}
```

```
    }
}

// The attempt to compile the example displays the following output:
//      error CS0453: The type 'T' must be a non-nullable value type in
//          order to use it as parameter 'T' in the generic type or
// method 'Number<T>'
```

The example compiles successfully if the constraint is added to the `FloatingPoint<T>` class.

C#

```
using System;

[assembly:CLSCCompliant(true)]

public class Number<T> where T : struct
{
    // use Double as the underlying type, since its range is a superset of
    // the ranges of all numeric types except BigInteger.
    protected double number;

    public Number(T value)
    {
        try {
            this.number = Convert.ToDouble(value);
        }
        catch (OverflowException e) {
            throw new ArgumentException("value is too large.", e);
        }
        catch (InvalidCastException e) {
            throw new ArgumentException("The value parameter is not numeric.",
e);
        }
    }

    public T Add(T value)
    {
        return (T) Convert.ChangeType(number + Convert.ToDouble(value),
typeof(T));
    }

    public T Subtract(T value)
    {
        return (T) Convert.ChangeType(number - Convert.ToDouble(value),
typeof(T));
    }
}

public class FloatingPoint<T> : Number<T> where T : struct
{
    public FloatingPoint(T number) : base(number)
    {
```

```
        if (typeof(float) == number.GetType() ||
            typeof(double) == number.GetType() ||
            typeof(decimal) == number.GetType())
                this.number = Convert.ToDouble(number);
            else
                throw new ArgumentException("The number parameter is not a
floating-point number.");
    }
}
```

The Common Language Specification imposes a conservative per-instantiation model for nested types and protected members. Open generic types cannot expose fields or members with signatures that contain a specific instantiation of a nested, protected generic type. Non-generic types that extend a specific instantiation of a generic base class or interface cannot expose fields or members with signatures that contain a different instantiation of a nested, protected generic type.

The following example defines a generic type, `C1<T>` (or `C1(of T)` in Visual Basic), and a protected class, `C1<T>.N` (or `C1(of T).N` in Visual Basic). `C1<T>` has two methods, `M1` and `M2`. However, `M1` is not CLS-compliant because it tries to return a `C1<int>.N` (or `C1(of Integer).N`) object from `C1<T>` (or `C1(of T)`). A second class, `C2`, is derived from `C1<long>` (or `C1(of Long)`). It has two methods, `M3` and `M4`. `M3` is not CLS-compliant because it tries to return a `C1<int>.N` (or `C1(of Integer).N`) object from a subclass of `C1<long>`. Language compilers can be even more restrictive. In this example, Visual Basic displays an error when it tries to compile `M4`.

C#

```

C1<long>

protected void M4(C1<long>.N n) { } // CLS-compliant, C1<long>.N is
                                    // accessible in C2 (extends
C1<long>
}
// Attempting to compile the example displays output like the following:
//      Generics4.cs(9,22): warning CS3001: Argument type 'C1<int>.N' is
not CLS-compliant
//      Generics4.cs(18,22): warning CS3001: Argument type 'C1<int>.N' is
not CLS-compliant

```

Constructors

Constructors in CLS-compliant classes and structures must follow these rules:

- A constructor of a derived class must call the instance constructor of its base class before it accesses inherited instance data. This requirement is because base class constructors are not inherited by their derived classes. This rule does not apply to structures, which do not support direct inheritance.

Typically, compilers enforce this rule independently of CLS compliance, as the following example shows. It creates a `Doctor` class that is derived from a `Person` class, but the `Doctor` class fails to call the `Person` class constructor to initialize inherited instance fields.

```

C#

using System;

[assembly: CLSCompliant(true)]

public class Person
{
    private string fName, lName, _id;

    public Person(string firstName, string lastName, string id)
    {
        if (String.IsNullOrEmpty(firstName + lastName))
            throw new ArgumentNullException("Either a first name or a last
name must be provided.");

        fName = firstName;
        lName = lastName;
        _id = id;
    }

    public string FirstName
    {

```

```

        get { return fName; }

    }

    public string LastName
    {
        get { return lName; }
    }

    public string Id
    {
        get { return _id; }
    }

    public override string ToString()
    {
        return String.Format("{0}{1}{2}", fName,
            String.IsNullOrEmpty(fName) ? "" : " ",
            lName);
    }
}

public class Doctor : Person
{
    public Doctor(string firstName, string lastName, string id)
    {
    }

    public override string ToString()
    {
        return "Dr. " + base.ToString();
    }
}

// Attempting to compile the example displays output like the
// following:
//   ctor1.cs(45,11): error CS1729: 'Person' does not contain a
// constructor that takes 0
//           arguments
//   ctor1.cs(10,11): (Location of symbol related to previous error)

```

- An object constructor cannot be called except to create an object. In addition, an object cannot be initialized twice. For example, this means that `Object.MemberwiseClone` and deserialization methods must not call constructors.

Properties

Properties in CLS-compliant types must follow these rules:

- A property must have a setter, a getter, or both. In an assembly, these are implemented as special methods, which means that they will appear as separate methods (the getter is named `get_propertyname` and the setter is

`set_propertyname)` marked as `SpecialName` in the assembly's metadata. The C# and Visual Basic compilers enforce this rule automatically without the need to apply the [CLSCompliantAttribute](#) attribute.

- A property's type is the return type of the property getter and the last argument of the setter. These types must be CLS compliant, and arguments cannot be assigned to the property by reference (that is, they cannot be managed pointers).
- If a property has both a getter and a setter, they must both be virtual, both static, or both instance. The C# and Visual Basic compilers automatically enforce this rule through their property definition syntax.

Events

An event is defined by its name and its type. The event type is a delegate that is used to indicate the event. For example, the [AppDomain.AssemblyResolve](#) event is of type [ResolveEventHandler](#). In addition to the event itself, three methods with names based on the event name provide the event's implementation and are marked as `SpecialName` in the assembly's metadata:

- A method for adding an event handler, named `add_EventName`. For example, the event subscription method for the [AppDomain.AssemblyResolve](#) event is named `add_AssemblyResolve`.
- A method for removing an event handler, named `remove_EventName`. For example, the removal method for the [AppDomain.AssemblyResolve](#) event is named `remove_AssemblyResolve`.
- A method for indicating that the event has occurred, named `raise_EventName`.

ⓘ Note

Most of the Common Language Specification's rules regarding events are implemented by language compilers and are transparent to component developers.

The methods for adding, removing, and raising the event must have the same accessibility. They must also all be static, instance, or virtual. The methods for adding and removing an event have one parameter whose type is the event delegate type. The add and remove methods must both be present or both be absent.

The following example defines a CLS-compliant class named `Temperature` that raises a `TemperatureChanged` event if the change in temperature between two readings equals or exceeds a threshold value. The `Temperature` class explicitly defines a `raise_TemperatureChanged` method so that it can selectively execute event handlers.

C#

```
using System;
using System.Collections;
using System.Collections.Generic;

[assembly: CLSCompliant(true)]

public class TemperatureChangedEventArgs : EventArgs
{
    private Decimal originalTemp;
    private Decimal newTemp;
    private DateTimeOffset when;

    public TemperatureChangedEventArgs(Decimal original, Decimal @new,
DateTimeOffset time)
    {
        originalTemp = original;
        newTemp = @new;
        when = time;
    }

    public Decimal OldTemperature
    {
        get { return originalTemp; }
    }

    public Decimal CurrentTemperature
    {
        get { return newTemp; }
    }

    public DateTimeOffset Time
    {
        get { return when; }
    }
}

public delegate void TemperatureChanged(Object sender,
TemperatureChangedEventArgs e);

public class Temperature
{
    private struct TemperatureInfo
    {
        public Decimal Temperature;
        public DateTimeOffset Recorded;
    }
```

```

public event TemperatureChanged TemperatureChanged;

private Decimal previous;
private Decimal current;
private Decimal tolerance;
private List<TemperatureInfo> tis = new List<TemperatureInfo>();

public Temperature(Decimal temperature, Decimal tolerance)
{
    current = temperature;
    TemperatureInfo ti = new TemperatureInfo();
    ti.Temperature = temperature;
    tis.Add(ti);
    ti.Recorded = DateTimeOffset.UtcNow;
    this.tolerance = tolerance;
}

public Decimal CurrentTemperature
{
    get { return current; }
    set {
        TemperatureInfo ti = new TemperatureInfo();
        ti.Temperature = value;
        ti.Recorded = DateTimeOffset.UtcNow;
        previous = current;
        current = value;
        if (Math.Abs(current - previous) >= tolerance)
            raise_TemperatureChanged(new
TemperatureChangedEventArgs(previous, current, ti.Recorded));
    }
}

public void raise_TemperatureChanged(TemperatureChangedEventArgs
EventArgs)
{
    if (TemperatureChanged == null)
        return;

    foreach (TemperatureChanged d in
TemperatureChanged.GetInvocationList()) {
        if (d.Method.Name.Contains("Duplicate"))
            Console.WriteLine("Duplicate event handler; event handler not
executed.");
        else
            d.Invoke(this, EventArgs);
    }
}

public class Example
{
    public Temperature temp;

    public static void Main()

```

```

{
    Example ex = new Example();
}

public Example()
{
    temp = new Temperature(65, 3);
    temp.TemperatureChanged += this.TemperatureNotification;
    RecordTemperatures();
    Example ex = new Example(temp);
    ex.RecordTemperatures();
}

public Example(Temperature t)
{
    temp = t;
    RecordTemperatures();
}

public void RecordTemperatures()
{
    temp.TemperatureChanged += this.DuplicateTemperatureNotification;
    temp.CurrentTemperature = 66;
    temp.CurrentTemperature = 63;
}

internal void TemperatureNotification(Object sender,
TemperatureChangedEventArgs e)
{
    Console.WriteLine("Notification 1: The temperature changed from {0} to
{1}", e.OldTemperature, e.CurrentTemperature);
}

public void DuplicateTemperatureNotification(Object sender,
TemperatureChangedEventArgs e)
{
    Console.WriteLine("Notification 2: The temperature changed from {0} to
{1}", e.OldTemperature, e.CurrentTemperature);
}
}

```

Overloads

The Common Language Specification imposes the following requirements on overloaded members:

- Members can be overloaded based on the number of parameters and the type of any parameter. Calling convention, return type, custom modifiers applied to the method or its parameter, and whether parameters are passed by value or by reference are not considered when differentiating between overloads. For an

example, see the code for the requirement that names must be unique within a scope in the [Naming conventions](#) section.

- Only properties and methods can be overloaded. Fields and events cannot be overloaded.
- Generic methods can be overloaded based on the number of their generic parameters.

ⓘ Note

The `op_Explicit` and `op_Implicit` operators are exceptions to the rule that return value is not considered part of a method signature for overload resolution. These two operators can be overloaded based on both their parameters and their return value.

Exceptions

Exception objects must derive from [System.Exception](#) or from another type derived from [System.Exception](#). The following example illustrates the compiler error that results when a custom class named `ErrorClass` is used for exception handling.

C#

```
using System;

[assembly: CLSCompliant(true)]

public class ErrorClass
{
    string msg;

    public ErrorClass(string errorMessage)
    {
        msg = errorMessage;
    }

    public string Message
    {
        get { return msg; }
    }
}

public static class StringUtils
{
    public static string[] SplitString(this string value, int index)
```

```

        if (index < 0 | index > value.Length) {
            ErrorClass badIndex = new ErrorClass("The index is not within the
string.");
            throw badIndex;
        }
        string[] retVal = { value.Substring(0, index - 1),
                            value.Substring(index) };
        return retVal;
    }
}
// Compilation produces a compiler error like the following:
// Exceptions1.cs(26,16): error CS0155: The type caught or thrown must be
derived from
//           System.Exception

```

To correct this error, the `ErrorClass` class must inherit from `System.Exception`. In addition, the `Message` property must be overridden. The following example corrects these errors to define an `ErrorClass` class that is CLS-compliant.

C#

```

using System;

[assembly: CLSCompliant(true)]

public class ErrorClass : Exception
{
    string msg;

    public ErrorClass(string errorMessage)
    {
        msg = errorMessage;
    }

    public override string Message
    {
        get { return msg; }
    }
}

public static class StringUtils
{
    public static string[] SplitString(this string value, int index)
    {
        if (index < 0 | index > value.Length) {
            ErrorClass badIndex = new ErrorClass("The index is not within the
string.");
            throw badIndex;
        }
        string[] retVal = { value.Substring(0, index - 1),
                            value.Substring(index) };
        return retVal;
    }
}

```

```
}
```

Attributes

In .NET assemblies, custom attributes provide an extensible mechanism for storing custom attributes and retrieving metadata about programming objects, such as assemblies, types, members, and method parameters. Custom attributes must derive from [System.Attribute](#) or from a type derived from [System.Attribute](#).

The following example violates this rule. It defines a `NumericAttribute` class that does not derive from [System.Attribute](#). A compiler error results only when the non-CLS-compliant attribute is applied, not when the class is defined.

C#

```
using System;

[assembly: CLSCompliant(true)]

[AttributeUsageAttribute(AttributeTargets.Class | AttributeTargets.Struct)]
public class NumericAttribute
{
    private bool _isNumeric;

    public NumericAttribute(bool isNumeric)
    {
        _isNumeric = isNumeric;
    }

    public bool IsNumeric
    {
        get { return _isNumeric; }
    }
}

[Numeric(true)] public struct UDouble
{
    double Value;
}
// Compilation produces a compiler error like the following:
//     Attribute1.cs(22,2): error CS0616: 'NumericAttribute' is not an
//     attribute class
//     Attribute1.cs(7,14): (Location of symbol related to previous error)
```

The constructor or the properties of a CLS-compliant attribute can expose only the following types:

- Boolean
- Byte
- Char
- Double
- Int16
- Int32
- Int64
- Single
- String
- Type
- Any enumeration type whose underlying type is Byte, Int16, Int32, or Int64.

The following example defines a `DescriptionAttribute` class that derives from `Attribute`. The class constructor has a parameter of type `Descriptor`, so the class is not CLS-compliant. The C# compiler emits a warning but compiles successfully, whereas the Visual Basic compiler doesn't emit a warning or an error.

C#

```
using System;

[assembly:CLSClaimedAttribute(true)]

public enum DescriptorType { type, member };

public class Descriptor
{
    public DescriptorType Type;
    public String Description;
}

[AttributeUsage(AttributeTargets.All)]
public class DescriptionAttribute : Attribute
{
    private Descriptor desc;

    public DescriptionAttribute(Descriptor d)
    {
        desc = d;
    }
}
```

```
public Descriptor Descriptor
{ get { return desc; } }
}
// Attempting to compile the example displays output like the following:
//     warning CS3015: 'DescriptionAttribute' has no accessible
//                     constructors which use only CLS-compliant types
```

The `CLSCompliantAttribute` attribute

The `CLSCompliantAttribute` attribute is used to indicate whether a program element complies with the Common Language Specification. The `CLSCompliantAttribute(Boolean)` constructor includes a single required parameter, `isCompliant`, that indicates whether the program element is CLS-compliant.

At compile time, the compiler detects non-compliant elements that are presumed to be CLS-compliant and emits a warning. The compiler does not emit warnings for types or members that are explicitly declared to be non-compliant.

Component developers can use the `CLSCompliantAttribute` attribute in two ways:

- To define the parts of the public interface exposed by a component that are CLS-compliant and the parts that are not CLS-compliant. When the attribute is used to mark particular program elements as CLS-compliant, its use guarantees that those elements are accessible from all languages and tools that target .NET.
- To ensure that the component library's public interface exposes only program elements that are CLS-compliant. If elements are not CLS-compliant, compilers will generally issue a warning.

⚠ Warning

In some cases, language compilers enforce CLS-compliant rules regardless of whether the `CLSCompliantAttribute` attribute is used. For example, defining a static member in an interface violates a CLS rule. In this regard, if you define a `static` (in C#) or `Shared` (in Visual Basic) member in an interface, both the C# and Visual Basic compilers display an error message and fail to compile the app.

The `CLSCompliantAttribute` attribute is marked with an `AttributeUsageAttribute` attribute that has a value of `AttributeTargets.All`. This value allows you to apply the `CLSCompliantAttribute` attribute to any program element, including assemblies, modules, types (classes, structures, enumerations, interfaces, and delegates), type members (constructors, methods, properties, fields, and events), parameters, generic

parameters, and return values. However, in practice, you should apply the attribute only to assemblies, types, and type members. Otherwise, compilers ignore the attribute and continue to generate compiler warnings whenever they encounter a non-compliant parameter, generic parameter, or return value in your library's public interface.

The value of the [CLSCompliantAttribute](#) attribute is inherited by contained program elements. For example, if an assembly is marked as CLS-compliant, its types are also CLS-compliant. If a type is marked as CLS-compliant, its nested types and members are also CLS-compliant.

You can explicitly override the inherited compliance by applying the [CLSCompliantAttribute](#) attribute to a contained program element. For example, you can use the [CLSCompliantAttribute](#) attribute with an `isCompliant` value of `false` to define a non-compliant type in a compliant assembly, and you can use the attribute with an `isCompliant` value of `true` to define a compliant type in a non-compliant assembly. You can also define non-compliant members in a compliant type. However, a non-compliant type cannot have compliant members, so you cannot use the attribute with an `isCompliant` value of `true` to override inheritance from a non-compliant type.

When you are developing components, you should always use the [CLSCompliantAttribute](#) attribute to indicate whether your assembly, its types, and its members are CLS-compliant.

To create CLS-compliant components:

1. Use the [CLSCompliantAttribute](#) to mark your assembly as CLS-compliant.
2. Mark any publicly exposed types in the assembly that are not CLS-compliant as non-compliant.
3. Mark any publicly exposed members in CLS-compliant types as non-compliant.
4. Provide a CLS-compliant alternative for non-CLS-compliant members.

If you've successfully marked all your non-compliant types and members, your compiler should not emit any non-compliance warnings. However, you should indicate which members are not CLS-compliant and list their CLS-compliant alternatives in your product documentation.

The following example uses the [CLSCompliantAttribute](#) attribute to define a CLS-compliant assembly and a type, `CharacterUtilities`, that has two non-CLS-compliant members. Because both members are tagged with the `CLSCompliant(false)` attribute, the compiler produces no warnings. The class also provides a CLS-compliant alternative for both methods. Ordinarily, we would just add two overloads to the `ToUTF16` method

to provide CLS-compliant alternatives. However, because methods cannot be overloaded based on return value, the names of the CLS-compliant methods are different from the names of the non-compliant methods.

C#

```
using System;
using System.Text;

[assembly:CLSCompliant(true)]

public class CharacterUtilities
{
    [CLSCompliant(false)] public static ushort ToUTF16(String s)
    {
        s = s.Normalize(NormalizationForm.FormC);
        return Convert.ToInt16(s[0]);
    }

    [CLSCompliant(false)] public static ushort ToUTF16(Char ch)
    {
        return Convert.ToInt16(ch);
    }

    // CLS-compliant alternative for ToUTF16(String).
    public static int ToUTF16CodeUnit(String s)
    {
        s = s.Normalize(NormalizationForm.FormC);
        return (int) Convert.ToInt16(s[0]);
    }

    // CLS-compliant alternative for ToUTF16(Char).
    public static int ToUTF16CodeUnit(Char ch)
    {
        return Convert.ToInt32(ch);
    }

    public bool HasMultipleRepresentations(String s)
    {
        String s1 = s.Normalize(NormalizationForm.FormC);
        return s.Equals(s1);
    }

    public int GetUnicodeCodePoint(Char ch)
    {
        if (Char.IsSurrogate(ch))
            throw new ArgumentException("ch cannot be a high or low
surrogate.");

        return Char.ConvertToUtf32(ch.ToString(), 0);
    }

    public int GetUnicodeCodePoint(Char[] chars)
```

```

{
    if (chars.Length > 2)
        throw new ArgumentException("The array has too many characters.");

    if (chars.Length == 2) {
        if (! Char.IsSurrogatePair(chars[0], chars[1]))
            throw new ArgumentException("The array must contain a low and a
high surrogate.");
        else
            return Char.ConvertToUtf32(chars[0], chars[1]);
    }
    else {
        return Char.ConvertToUtf32(chars.ToString(), 0);
    }
}

```

If you are developing an app rather than a library (that is, if you aren't exposing types or members that can be consumed by other app developers), the CLS compliance of the program elements that your app consumes are of interest only if your language does not support them. In that case, your language compiler will generate an error when you try to use a non-CLS-compliant element.

Cross-language interoperability

Language independence has a few possible meanings. One meaning involves seamlessly consuming types written in one language from an app written in another language. A second meaning, which is the focus of this article, involves combining code written in multiple languages into a single .NET assembly.

The following example illustrates cross-language interoperability by creating a class library named Utilities.dll that includes two classes, `NumericLib` and `StringLib`. The `NumericLib` class is written in C#, and the `StringLib` class is written in Visual Basic. Here's the source code for `StringUtil.vb`, which includes a single member, `ToTitleCase`, in its `StringLib` class.

VB

```

Imports System.Collections.Generic
Imports System.Runtime.CompilerServices

Public Module StringLib
    Private exclusions As List(Of String)

    Sub New()
        Dim words() As String = {"a", "an", "and", "of", "the"}
        exclusions = New List(Of String)

```

```

        exclusions.AddRange(words)
    End Sub

    <Extension()> _
    Public Function ToTitleCase(title As String) As String
        Dim words() As String = title.Split()
        Dim result As String = String.Empty

        For ctr As Integer = 0 To words.Length - 1
            Dim word As String = words(ctr)
            If ctr = 0 OrElse Not exclusions.Contains(word.ToLower()) Then
                result += word.Substring(0, 1).ToUpper() + _
                          word.Substring(1).ToLower()
            Else
                result += word.ToLower()
            End If
            If ctr <= words.Length - 1 Then
                result += " "
            End If
        Next
        Return result
    End Function
End Module

```

Here's the source code for NumberUtil.cs, which defines a `NumericLib` class that has two members, `IsEven` and `NearZero`.

C#

```

using System;

public static class NumericLib
{
    public static bool IsEven(this IConvertible number)
    {
        if (number is Byte ||
            number is SByte ||
            number is Int16 ||
            number is UInt16 ||
            number is Int32 ||
            number is UInt32 ||
            number is Int64)
            return Convert.ToInt64(number) % 2 == 0;
        else if (number is UInt64)
            return ((ulong) number) % 2 == 0;
        else
            throw new NotSupportedException("IsEven called for a non-integer
value.");
    }

    public static bool NearZero(double number)
    {
        return Math.Abs(number) < .00001;
    }
}

```

```
}
```

To package the two classes in a single assembly, you must compile them into modules. To compile the Visual Basic source code file into a module, use this command:

Console

```
vbc /t:module StringUtil.vb
```

For more information about the command-line syntax of the Visual Basic compiler, see [Building from the Command Line](#).

To compile the C# source code file into a module, use this command:

Console

```
csc /t:module NumberUtil.cs
```

You then use the [Linker options](#) to compile the two modules into an assembly:

Console

```
link numberutil.netmodule stringutil.netmodule /out:UtilityLib.dll /dll
```

The following example then calls the `NumericLib.NearZero` and `StringLib.ToTitleCase` methods. Both the Visual Basic code and the C# code are able to access the methods in both classes.

C#

```
using System;

public class Example
{
    public static void Main()
    {
        Double dbl = 0.0 - Double.Epsilon;
        Console.WriteLine(NumericLib.NearZero(dbl));

        string s = "war and peace";
        Console.WriteLine(s.ToTitleCase());
    }
}

// The example displays the following output:
```

```
//      True  
//      War and Peace
```

To compile the Visual Basic code, use this command:

Console

```
vbc example.vb /r:UtilityLib.dll
```

To compile with C#, change the name of the compiler from `vbc` to `csc`, and change the file extension from `.vb` to `.cs`:

Console

```
csc example.cs /r:UtilityLib.dll
```

 **Collaborate with us on GitHub**

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

 .NET

.NET feedback

.NET is an open source project.
Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

Type conversion in .NET

Article • 09/15/2021

Every value has an associated type, which defines attributes such as the amount of space allocated to the value, the range of possible values it can have, and the members that it makes available. Many values can be expressed as more than one type. For example, the value 4 can be expressed as an integer or a floating-point value. Type conversion creates a value in a new type that is equivalent to the value of an old type, but does not necessarily preserve the identity (or exact value) of the original object.

.NET automatically supports the following conversions:

- Conversion from a derived class to a base class. This means, for example, that an instance of any class or structure can be converted to an [Object](#) instance. This conversion does not require a casting or conversion operator.
- Conversion from a base class back to the original derived class. In C#, this conversion requires a casting operator. In Visual Basic, it requires the `CType` operator if `Option Strict` is on.
- Conversion from a type that implements an interface to an interface object that represents that interface. This conversion does not require a casting or conversion operator.
- Conversion from an interface object back to the original type that implements that interface. In C#, this conversion requires a casting operator. In Visual Basic, it requires the `CType` operator if `Option Strict` is on.

In addition to these automatic conversions, .NET provides several features that support custom type conversion. These include the following:

- The `Implicit` operator, which defines the available widening conversions between types. For more information, see the [Implicit Conversion with the Implicit Operator](#) section.
- The `Explicit` operator, which defines the available narrowing conversions between types. For more information, see the [Explicit Conversion with the Explicit Operator](#) section.
- The [IConvertible](#) interface, which defines conversions to each of the base .NET data types. For more information, see [The IConvertible Interface](#) section.

- The [Convert](#) class, which provides a set of methods that implement the methods in the [IConvertible](#) interface. For more information, see [The Convert Class](#) section.
- The [TypeConverter](#) class, which is a base class that can be extended to support the conversion of a specified type to any other type. For more information, see [The TypeConverter Class](#) section.

Implicit conversion with the `implicit` operator

Widening conversions involve the creation of a new value from the value of an existing type that has either a more restrictive range or a more restricted member list than the target type. Widening conversions cannot result in data loss (although they may result in a loss of precision). Because data cannot be lost, compilers can handle the conversion implicitly or transparently, without requiring the use of an explicit conversion method or a casting operator.

 **Note**

Although code that performs an implicit conversion can call a conversion method or use a casting operator, their use is not required by compilers that support implicit conversions.

For example, the [Decimal](#) type supports implicit conversions from [Byte](#), [Char](#), [Int16](#), [Int32](#), [Int64](#), [SByte](#), [UInt16](#), [UInt32](#), and [UInt64](#) values. The following example illustrates some of these implicit conversions in assigning values to a [Decimal](#) variable.

C#

```
byte byteValue = 16;
short shortValue = -1024;
int intValue = -1034000;
long longValue = 1152921504606846976;
ulong ulongValue = UInt64.MaxValue;

decimal decimalValue;

decimalValue = byteValue;
Console.WriteLine("After assigning a {0} value, the Decimal value is
{1}.",
    byteValue.GetType().Name, decimalValue);

decimalValue = shortValue;
Console.WriteLine("After assigning a {0} value, the Decimal value is
{1}.",
    shortValue.GetType().Name, decimalValue);
```

```

decimalValue = intValue;
Console.WriteLine("After assigning a {0} value, the Decimal value is {1}.",
    intValue.GetType().Name, decimalValue);

decimalValue = longValue;
Console.WriteLine("After assigning a {0} value, the Decimal value is {1}.",
    longValue.GetType().Name, decimalValue);

decimalValue = ulongValue;
Console.WriteLine("After assigning a {0} value, the Decimal value is {1}.",
    ulongValue.GetType().Name, decimalValue);

// The example displays the following output:
//     After assigning a Byte value, the Decimal value is 16.
//     After assigning a Int16 value, the Decimal value is -1024.
//     After assigning a Int32 value, the Decimal value is -1034000.
//     After assigning a Int64 value, the Decimal value is
1152921504606846976.
//     After assigning a UInt64 value, the Decimal value is
18446744073709551615.

```

If a particular language compiler supports custom operators, you can also define implicit conversions in your own custom types. The following example provides a partial implementation of a signed byte data type named `ByteWithSign` that uses sign-and-magnitude representation. It supports implicit conversion of `Byte` and `SByte` values to `ByteWithSign` values.

C#

```

public struct ByteWithSign
{
    private SByte signValue;
    private Byte value;

    public static implicit operator ByteWithSign(SByte value)
    {
        ByteWithSign newValue;
        newValue.signValue = (SByte)Math.Sign(value);
        newValue.value = (byte)Math.Abs(value);
        return newValue;
    }

    public static implicit operator ByteWithSign(Byte value)
    {
        ByteWithSign newValue;
        newValue.signValue = 1;
        newValue.value = value;
        return newValue;
    }
}

```

```
public override string ToString()
{
    return (signValue * value).ToString();
}
```

Client code can then declare a `ByteWithSign` variable and assign it `Byte` and `SByte` values without performing any explicit conversions or using any casting operators, as the following example shows.

C#

```
SByte sbyteValue = -120;
ByteWithSign value = sbyteValue;
Console.WriteLine(value);
value = Byte.MaxValue;
Console.WriteLine(value);
// The example displays the following output:
//      -120
//      255
```

Explicit conversion with the `explicit` operator

Narrowing conversions involve the creation of a new value from the value of an existing type that has either a greater range or a larger member list than the target type. Because a narrowing conversion can result in a loss of data, compilers often require that the conversion be made explicit through a call to a conversion method or a casting operator. That is, the conversion must be handled explicitly in developer code.

ⓘ Note

The major purpose of requiring a conversion method or casting operator for narrowing conversions is to make the developer aware of the possibility of data loss or an `OverflowException` so that it can be handled in code. However, some compilers can relax this requirement. For example, in Visual Basic, if `Option Strict` is off (its default setting), the Visual Basic compiler tries to perform narrowing conversions implicitly.

For example, the `UInt32`, `Int64`, and `UInt64` data types have ranges that exceed that the `Int32` data type, as the following table shows.

| Type | Comparison with range of Int32 |
|--------|--|
| Int64 | Int64.MaxValue is greater than Int32.MaxValue, and Int64.MinValue is less than (has a greater negative range than) Int32.MinValue. |
| UInt32 | UInt32.MaxValue is greater than Int32.MaxValue. |
| UInt64 | UInt64.MaxValue is greater than Int32.MaxValue. |

To handle such narrowing conversions, .NET allows types to define an `Explicit` operator. Individual language compilers can then implement this operator using their own syntax, or a member of the `Convert` class can be called to perform the conversion. (For more information about the `Convert` class, see [The Convert Class](#) later in this topic.) The following example illustrates the use of language features to handle the explicit conversion of these potentially out-of-range integer values to `Int32` values.

C#

```
long number1 = int.MaxValue + 20L;
uint number2 = int.MaxValue - 1000;
ulong number3 = int.MaxValue;

int intNumber;

try
{
    intNumber = checked((int)number1);
    Console.WriteLine("After assigning a {0} value, the Integer value is
{1}.",
                      number1.GetType().Name, intNumber);
}
catch (OverflowException)
{
    if (number1 > int.MaxValue)
        Console.WriteLine("Conversion failed: {0} exceeds {1}.",
                          number1, int.MaxValue);
    else
        Console.WriteLine("Conversion failed: {0} is less than {1}.",
                          number1, int.MinValue);
}

try
{
    intNumber = checked((int)number2);
    Console.WriteLine("After assigning a {0} value, the Integer value is
{1}.",
                      number2.GetType().Name, intNumber);
}
catch (OverflowException)
{
    Console.WriteLine("Conversion failed: {0} exceeds {1}.",
                      number2, int.MaxValue);
}
```

```

        number2, int.MaxValue);
}

try
{
    intNumber = checked((int)number3);
    Console.WriteLine("After assigning a {0} value, the Integer value is
{1}.",
                      number3.GetType().Name, intNumber);
}
catch (OverflowException)
{
    Console.WriteLine("Conversion failed: {0} exceeds {1}.",
                      number1, int.MaxValue);
}

// The example displays the following output:
//      Conversion failed: 2147483667 exceeds 2147483647.
//      After assigning a UInt32 value, the Integer value is 2147482647.
//      After assigning a UInt64 value, the Integer value is 2147483647.

```

Explicit conversions can produce different results in different languages, and these results can differ from the value returned by the corresponding [Convert](#) method. For example, if the [Double](#) value 12.63251 is converted to an [Int32](#), both the Visual Basic [CInt](#) method and the .NET [Convert.ToInt32\(Double\)](#) method round the [Double](#) to return a value of 13, but the C# [\(int\)](#) operator truncates the [Double](#) to return a value of 12. Similarly, the C# [\(int\)](#) operator does not support Boolean-to-integer conversion, but the Visual Basic [CInt](#) method converts a value of [true](#) to -1. On the other hand, the [Convert.ToInt32\(Boolean\)](#) method converts a value of [true](#) to 1.

Most compilers allow explicit conversions to be performed in a checked or unchecked manner. When a checked conversion is performed, an [OverflowException](#) is thrown when the value of the type to be converted is outside the range of the target type. When an unchecked conversion is performed under the same conditions, the conversion might not throw an exception, but the exact behavior becomes undefined and an incorrect value might result.

Note

In C#, checked conversions can be performed by using the [checked](#) keyword together with a casting operator, or by specifying the [/checked+](#) compiler option. Conversely, unchecked conversions can be performed by using the [unchecked](#) keyword together with the casting operator, or by specifying the [/checked-](#) compiler option. By default, explicit conversions are unchecked. In Visual Basic, checked conversions can be performed by clearing the [Remove integer overflow](#)

checks check box in the project's **Advanced Compiler Settings** dialog box, or by specifying the `/removeintchecks-` compiler option. Conversely, unchecked conversions can be performed by selecting the **Remove integer overflow checks** check box in the project's **Advanced Compiler Settings** dialog box or by specifying the `/removeintchecks+` compiler option. By default, explicit conversions are checked.

The following C# example uses the `checked` and `unchecked` keywords to illustrate the difference in behavior when a value outside the range of a `Byte` is converted to a `Byte`. The checked conversion throws an exception, but the unchecked conversion assigns `Byte.MaxValue` to the `Byte` variable.

C#

```
int largeValue = Int32.MaxValue;
byte newValue;

try
{
    newValue = unchecked((byte)largeValue);
    Console.WriteLine("Converted the {0} value {1} to the {2} value {3}.",
                      largeValue.GetType().Name, largeValue,
                      newValue.GetType().Name, newValue);
}
catch (OverflowException)
{
    Console.WriteLine("{0} is outside the range of the Byte data type.",
                      largeValue);
}

try
{
    newValue = checked((byte)largeValue);
    Console.WriteLine("Converted the {0} value {1} to the {2} value {3}.",
                      largeValue.GetType().Name, largeValue,
                      newValue.GetType().Name, newValue);
}
catch (OverflowException)
{
    Console.WriteLine("{0} is outside the range of the Byte data type.",
                      largeValue);
}

// The example displays the following output:
//     Converted the Int32 value 2147483647 to the Byte value 255.
//     2147483647 is outside the range of the Byte data type.
```

If a particular language compiler supports custom overloaded operators, you can also define explicit conversions in your own custom types. The following example provides a

partial implementation of a signed byte data type named `ByteWithSign` that uses sign-and-magnitude representation. It supports explicit conversion of `Int32` and `UInt32` values to `ByteWithSign` values.

C#

```
public struct ByteWithSignE
{
    private SByte signValue;
    private Byte value;

    private const byte.MaxValue = byte.MaxValue;
    private const int.MinValue = -1 * byte.MaxValue;

    public static explicit operator ByteWithSignE(int value)
    {
        // Check for overflow.
        if (value > ByteWithSignE.MaxValue || value <
ByteWithSignE.MinValue)
            throw new OverflowException(String.Format("{0}' is out of range
of the ByteWithSignE data type.",
                                            value));

        ByteWithSignE newValue;
        newValue.signValue = (SByte)Math.Sign(value);
        newValue.value = (byte)Math.Abs(value);
        return newValue;
    }

    public static explicit operator ByteWithSignE(uint value)
    {
        if (value > ByteWithSignE.MaxValue)
            throw new OverflowException(String.Format("{0}' is out of range
of the ByteWithSignE data type.",
                                            value));

        ByteWithSignE newValue;
        newValue.signValue = 1;
        newValue.value = (byte)value;
        return newValue;
    }

    public override string ToString()
    {
        return (signValue * value).ToString();
    }
}
```

Client code can then declare a `ByteWithSign` variable and assign it `Int32` and `UInt32` values if the assignments include a casting operator or a conversion method, as the following example shows.

C#

```
ByteWithSignE value;

try
{
    int intValue = -120;
    value = (ByteWithSignE)intValue;
    Console.WriteLine(value);
}
catch (OverflowException e)
{
    Console.WriteLine(e.Message);
}

try
{
    uint uintValue = 1024;
    value = (ByteWithSignE)uintValue;
    Console.WriteLine(value);
}
catch (OverflowException e)
{
    Console.WriteLine(e.Message);
}

// The example displays the following output:
//      -120
//      '1024' is out of range of the ByteWithSignE data type.
```

The [IConvertible](#) interface

To support the conversion of any type to a common language runtime base type, .NET provides the [IConvertible](#) interface. The implementing type is required to provide the following:

- A method that returns the [TypeCode](#) of the implementing type.
- Methods to convert the implementing type to each common language runtime base type ([Boolean](#), [Byte](#), [DateTime](#), [Decimal](#), [Double](#), and so on).
- A generalized conversion method to convert an instance of the implementing type to another specified type. Conversions that are not supported should throw an [InvalidOperationException](#).

Each common language runtime base type (that is, the [Boolean](#), [Byte](#), [Char](#), [DateTime](#), [Decimal](#), [Double](#), [Int16](#), [Int32](#), [Int64](#), [SByte](#), [Single](#), [String](#), [UInt16](#), [UInt32](#), and [UInt64](#)), as well as the [DBNull](#) and [Enum](#) types, implement the [IConvertible](#) interface. However, these are explicit interface implementations; the conversion method can be called only

through an [IConvertible](#) interface variable, as the following example shows. This example converts an [Int32](#) value to its equivalent [Char](#) value.

C#

```
int codePoint = 1067;
IConvertible iConv = codePoint;
char ch = iConv.ToChar(null);
Console.WriteLine("Converted {0} to {1}.", codePoint, ch);
```

The requirement to call the conversion method on its interface rather than on the implementing type makes explicit interface implementations relatively expensive. Instead, we recommend that you call the appropriate member of the [Convert](#) class to convert between common language runtime base types. For more information, see the next section, [The Convert Class](#).

ⓘ Note

In addition to the [IConvertible](#) interface and the [Convert](#) class provided by .NET, individual languages may also provide ways to perform conversions. For example, C# uses casting operators; Visual Basic uses compiler-implemented conversion functions such as `CType`, `CInt`, and `DirectCast`.

For the most part, the [IConvertible](#) interface is designed to support conversion between the base types in .NET. However, the interface can also be implemented by a custom type to support conversion of that type to other custom types. For more information, see the section [Custom Conversions with the ChangeType Method](#) later in this topic.

The Convert class

Although each base type's [IConvertible](#) interface implementation can be called to perform a type conversion, calling the methods of the [System.Convert](#) class is the recommended language-neutral way to convert from one base type to another. In addition, the [Convert.ChangeType\(Object, Type, IFormatProvider\)](#) method can be used to convert from a specified custom type to another type.

Conversions between base types

The [Convert](#) class provides a language-neutral way to perform conversions between base types and is available to all languages that target the common language runtime. It provides a complete set of methods for both widening and narrowing conversions, and

throws an [InvalidCastException](#) for conversions that are not supported (such as the conversion of a [DateTime](#) value to an integer value). Narrowing conversions are performed in a checked context, and an [OverflowException](#) is thrown if the conversion fails.

ⓘ Important

Because the [Convert](#) class includes methods to convert to and from each base type, it eliminates the need to call each base type's [IConvertible](#) explicit interface implementation.

The following example illustrates the use of the [System.Convert](#) class to perform several widening and narrowing conversions between .NET base types.

C#

```
// Convert an Int32 value to a Decimal (a widening conversion).
int integralValue = 12534;
decimal decimalValue = Convert.ToDecimal(integralValue);
Console.WriteLine("Converted the {0} value {1} to " +
                  "the {2} value {3:N2}.",
                  integralValue.GetType().Name,
                  integralValue,
                  decimalValue.GetType().Name,
                  decimalValue);

// Convert a Byte value to an Int32 value (a widening conversion).
byte byteValue = Byte.MaxValue;
int integralValue2 = Convert.ToInt32(byteValue);
Console.WriteLine("Converted the {0} value {1} to " +
                  "the {2} value {3:G}.",
                  byteValue.GetType().Name,
                  byteValue,
                  integralValue2.GetType().Name,
                  integralValue2);

// Convert a Double value to an Int32 value (a narrowing conversion).
double doubleValue = 16.32513e12;
try
{
    long longValue = Convert.ToInt64(doubleValue);
    Console.WriteLine("Converted the {0} value {1:E} to " +
                      "the {2} value {3:N0}.",
                      doubleValue.GetType().Name,
                      doubleValue,
                      longValue.GetType().Name,
                      longValue);
}
catch (OverflowException)
{
    Console.WriteLine("Unable to convert the {0:E} value {1}.",
```

```

                doubleValue.GetType().Name,
doubleValue);
}

// Convert a signed byte to a byte (a narrowing conversion).
sbyte sbyteValue = -16;
try
{
    byte byteValue2 = Convert.ToByte(sbyteValue);
    Console.WriteLine("Converted the {0} value {1} to " +
                      "the {2} value {3:G}.",
                      sbyteValue.GetType().Name,
                      sbyteValue,
                      byteValue2.GetType().Name,
                      byteValue2);
}
catch (OverflowException)
{
    Console.WriteLine("Unable to convert the {0} value {1}.",
                      sbyteValue.GetType().Name,
                      sbyteValue);
}
// The example displays the following output:
//     Converted the Int32 value 12534 to the Decimal value 12,534.00.
//     Converted the Byte value 255 to the Int32 value 255.
//     Converted the Double value 1.632513E+013 to the Int64 value
16,325,130,000,000.
//     Unable to convert the SByte value -16.

```

In some cases, particularly when converting to and from floating-point values, a conversion may involve a loss of precision, even though it does not throw an [OverflowException](#). The following example illustrates this loss of precision. In the first case, a [Decimal](#) value has less precision (fewer significant digits) when it is converted to a [Double](#). In the second case, a [Double](#) value is rounded from 42.72 to 43 in order to complete the conversion.

C#

```

double doubleValue;

// Convert a Double to a Decimal.
decimal decimalValue = 13956810.96702888123451471211m;
doubleValue = Convert.ToDouble(decimalValue);
Console.WriteLine("{0} converted to {1}.", decimalValue, doubleValue);

doubleValue = 42.72;
try
{
    int integerValue = Convert.ToInt32(doubleValue);
    Console.WriteLine("{0} converted to {1}.",
                      doubleValue, integerValue);
}

```

```
        }
        catch (OverflowException)
        {
            Console.WriteLine("Unable to convert {0} to an integer.",
                doubleValue);
        }
    // The example displays the following output:
    //      13956810.96702888123451471211 converted to 13956810.9670289.
    //      42.72 converted to 43.
```

For a table that lists both the widening and narrowing conversions supported by the [Convert](#) class, see [Type Conversion Tables](#).

Custom conversions with the `ChangeType` method

In addition to supporting conversions to each of the base types, the [Convert](#) class can be used to convert a custom type to one or more predefined types. This conversion is performed by the [Convert.ChangeType\(Object, Type, IFormatProvider\)](#) method, which in turn wraps a call to the [IConvertible.ToType](#) method of the `value` parameter. This means that the object represented by the `value` parameter must provide an implementation of the [IConvertible](#) interface.

ⓘ Note

Because the [Convert.ChangeType\(Object, Type\)](#) and [Convert.ChangeType\(Object, Type, IFormatProvider\)](#) methods use a `Type` object to specify the target type to which `value` is converted, they can be used to perform a dynamic conversion to an object whose type is not known at compile time. However, note that the [IConvertible](#) implementation of `value` must still support this conversion.

The following example illustrates a possible implementation of the [IConvertible](#) interface that allows a `TemperatureCelsius` object to be converted to a `TemperatureFahrenheit` object and vice versa. The example defines a base class, `Temperature`, that implements the [IConvertible](#) interface and overrides the [Object.ToString](#) method. The derived `TemperatureCelsius` and `TemperatureFahrenheit` classes each override the `ToType` and the `ToString` methods of the base class.

C#

```
using System;

public abstract class Temperature : IConvertible
{
```

```
protected decimal temp;

public Temperature(decimal temperature)
{
    this.temp = temperature;
}

public decimal Value
{
    get { return this.temp; }
    set { this.temp = value; }
}

public override string ToString()
{
    return temp.ToString(null as IFormatProvider) + "°";
}

// IConvertible implementations.
public TypeCode GetTypeCode()
{
    return TypeCode.Object;
}

public bool ToBoolean(IFormatProvider provider)
{
    throw new InvalidCastException(String.Format("Temperature-to-Boolean conversion is not supported."));
}

public byte ToByte(IFormatProvider provider)
{
    if (temp < Byte.MinValue || temp > Byte.MaxValue)
        throw new OverflowException(String.Format("{0} is out of range of the Byte data type.", temp));
    else
        return (byte)temp;
}

public char ToChar(IFormatProvider provider)
{
    throw new InvalidCastException("Temperature-to-Char conversion is not supported.");
}

public DateTime ToDateTime(IFormatProvider provider)
{
    throw new InvalidCastException("Temperature-to-DateTime conversion is not supported.");
}

public decimal ToDecimal(IFormatProvider provider)
{
    return temp;
}
```

```
public double ToDouble(IFormatProvider provider)
{
    return (double)temp;
}

public shortToInt16(IFormatProvider provider)
{
    if (temp < Int16.MinValue || temp > Int16.MaxValue)
        throw new OverflowException(String.Format("{0} is out of range
of the Int16 data type.", temp));
    else
        return (short)Math.Round(temp);
}

public intToInt32(IFormatProvider provider)
{
    if (temp < Int32.MinValue || temp > Int32.MaxValue)
        throw new OverflowException(String.Format("{0} is out of range
of the Int32 data type.", temp));
    else
        return (int)Math.Round(temp);
}

public longToInt64(IFormatProvider provider)
{
    if (temp < Int64.MinValue || temp > Int64.MaxValue)
        throw new OverflowException(String.Format("{0} is out of range
of the Int64 data type.", temp));
    else
        return (long)Math.Round(temp);
}

public sbyteToSByte(IFormatProvider provider)
{
    if (temp < SByte.MinValue || temp > SByte.MaxValue)
        throw new OverflowException(String.Format("{0} is out of range
of the SByte data type.", temp));
    else
        return (sbyte)temp;
}

public floatToSingle(IFormatProvider provider)
{
    return (float)temp;
}

public virtual stringToString(IFormatProvider provider)
{
    return temp.ToString(provider) + "°";
}

// If conversionType is implemented by another IConvertible method, call
it.
public virtual objectToType(Type conversionType, IFormatProvider
```

```

provider)
{
    switch (Type.GetTypeCode(conversionType))
    {
        case TypeCode.Boolean:
            return this.ToBoolean(provider);
        case TypeCode.Byte:
            return this.ToByte(provider);
        case TypeCode.Char:
            return this.ToChar(provider);
        case TypeCode.DateTime:
            return this.ToDateTime(provider);
        case TypeCode.Decimal:
            return this.ToDecimal(provider);
        case TypeCode.Double:
            return this.ToDouble(provider);
        case TypeCode.Empty:
            throw new NullReferenceException("The target type is
null.");
        case TypeCode.Int16:
            return this.ToInt16(provider);
        case TypeCode.Int32:
            return this.ToInt32(provider);
        case TypeCode.Int64:
            return this.ToInt64(provider);
        case TypeCode.Object:
            // Leave conversion of non-base types to derived classes.
            throw new InvalidCastException(String.Format("Cannot convert
from Temperature to {0}.",
                                                conversionType.Name));
        case TypeCode.SByte:
            return this.ToSByte(provider);
        case TypeCode.Single:
            return this.ToSingle(provider);
        case TypeCode.String:
            IConvertible iconv = this;
            return iconv.ToString(provider);
        case TypeCode.UInt16:
            return this.ToUInt16(provider);
        case TypeCode.UInt32:
            return this.ToUInt32(provider);
        case TypeCode.UInt64:
            return this.ToUInt64(provider);
        default:
            throw new InvalidCastException("Conversion not supported.");
    }
}

public ushort ToUInt16(IFormatProvider provider)
{
    if (temp < UInt16.MinValue || temp > UInt16.MaxValue)
        throw new OverflowException(String.Format("{0} is out of range
of the UInt16 data type.", temp));
    else
        return (ushort)Math.Round(temp);
}

```

```

    }

    public uint ToUInt32(IFormatProvider provider)
    {
        if (temp < UInt32.MinValue || temp > UInt32.MaxValue)
            throw new OverflowException(String.Format("{0} is out of range
of the UInt32 data type.", temp));
        else
            return (uint)Math.Round(temp);
    }

    public ulong ToUInt64(IFormatProvider provider)
    {
        if (temp < UInt64.MinValue || temp > UInt64.MaxValue)
            throw new OverflowException(String.Format("{0} is out of range
of the UInt64 data type.", temp));
        else
            return (ulong)Math.Round(temp);
    }
}

public class TemperatureCelsius : Temperature, IConvertible
{
    public TemperatureCelsius(decimal value) : base(value)
    {
    }

    // Override ToString methods.
    public override string ToString()
    {
        return this.ToString(null);
    }

    public override string ToString(IFormatProvider provider)
    {
        return temp.ToString(provider) + "°C";
    }

    // If conversionType is a implemented by another IConvertible method,
    // call it.
    public override object ToType(Type conversionType, IFormatProvider
provider)
    {
        // For non-objects, call base method.
        if (Type.GetTypeCode(conversionType) != TypeCode.Object)
        {
            return base.ToType(conversionType, provider);
        }
        else
        {
            if (conversionType.Equals(typeof(TemperatureCelsius)))
                return this;
            else if (conversionType.Equals(typeof(TemperatureFahrenheit)))
                return new TemperatureFahrenheit((decimal)this.temp * 9 / 5
+ 32);
        }
    }
}

```

```

        else
            throw new InvalidCastException(String.Format("Cannot convert
from Temperature to {0}.",
                                            conversionType.Name));
    }
}

public class TemperatureFahrenheit : Temperature, IConvertible
{
    public TemperatureFahrenheit(decimal value) : base(value)
    {
    }

    // Override ToString methods.
    public override string ToString()
    {
        return this.ToString(null);
    }

    public override string ToString(IFormatProvider provider)
    {
        return temp.ToString(provider) + "°F";
    }

    public override object ToType(Type conversionType, IFormatProvider
provider)
    {
        // For non-objects, call base method.
        if (Type.GetTypeCode(conversionType) != TypeCode.Object)
        {
            return base.ToType(conversionType, provider);
        }
        else
        {
            // Handle conversion between derived classes.
            if (conversionType.Equals(typeof(TemperatureFahrenheit)))
                return this;
            else if (conversionType.Equals(typeof(TemperatureCelsius)))
                return new TemperatureCelsius((decimal)(this.temp - 32) * 5
/ 9);
            // Unspecified object type: throw an InvalidCastException.
            else
                throw new InvalidCastException(String.Format("Cannot convert
from Temperature to {0}.",
                                            conversionType.Name));
        }
    }
}

```

The following example illustrates several calls to these `IConvertible` implementations to convert `TemperatureCelsius` objects to `TemperatureFahrenheit` objects and vice versa.

C#

```
TemperatureCelsius tempC1 = new TemperatureCelsius(0);
TemperatureFahrenheit tempF1 =
(TemperatureFahrenheit)Convert.ChangeType(tempC1,
typeof(TemperatureFahrenheit), null);
Console.WriteLine("{0} equals {1}.", tempC1, tempF1);
TemperatureCelsius tempC2 = (TemperatureCelsius)Convert.ChangeType(tempC1,
typeof(TemperatureCelsius), null);
Console.WriteLine("{0} equals {1}.", tempC1, tempC2);
TemperatureFahrenheit tempF2 = new TemperatureFahrenheit(212);
TemperatureCelsius tempC3 = (TemperatureCelsius)Convert.ChangeType(tempF2,
typeof(TemperatureCelsius), null);
Console.WriteLine("{0} equals {1}.", tempF2, tempC3);
TemperatureFahrenheit tempF3 =
(TemperatureFahrenheit)Convert.ChangeType(tempF2,
typeof(TemperatureFahrenheit), null);
Console.WriteLine("{0} equals {1}.", tempF2, tempF3);
// The example displays the following output:
//      0°C equals 32°F.
//      0°C equals 0°C.
//      212°F equals 100°C.
//      212°F equals 212°F.
```

The TypeConverter class

.NET also allows you to define a type converter for a custom type by extending the [System.ComponentModel.TypeConverter](#) class and associating the type converter with the type through a [System.ComponentModel.TypeConverterAttribute](#) attribute. The following table highlights the differences between this approach and implementing the [IConvertible](#) interface for a custom type.

ⓘ Note

Design-time support can be provided for a custom type only if it has a type converter defined for it.

| Conversion using TypeConverter | Conversion using IConvertible |
|--|--|
| Is implemented for a custom type by deriving a separate class from TypeConverter . This derived class is associated with the custom type by applying a TypeConverterAttribute attribute. | Is implemented by a custom type to perform conversion. A user of the type invokes an IConvertible conversion method on the type. |
| Can be used both at design time and at run time. | Can be used only at run time. |

| Conversion using TypeConverter | Conversion using IConvertible |
|---|---|
| Uses reflection; therefore, is slower than conversion enabled by IConvertible . | Does not use reflection. |
| Allows two-way type conversions from the custom type to other data types, and from other data types to the custom type. For example, a TypeConverter defined for <code>MyType</code> allows conversions from <code>MyType</code> to String , and from String to <code>MyType</code> . | Allows conversion from a custom type to other data types, but not from other data types to the custom type. |

For more information about using type converters to perform conversions, see [System.ComponentModel.TypeConverter](#).

See also

- [System.Convert](#)
- [IConvertible](#)
- [Type Conversion Tables](#)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

Type conversion tables in .NET

Article • 09/15/2021

Widening conversion occurs when a value of one type is converted to another type that is of equal or greater size. A narrowing conversion occurs when a value of one type is converted to a value of another type that is of a smaller size. The tables in this topic illustrate the behaviors exhibited by both types of conversions.

Widening conversions

The following table describes the widening conversions that can be performed without the loss of information.

[] Expand table

| Type | Can be converted without data loss to |
|--------|--|
| Byte | UInt16, Int16, UInt32, Int32, UInt64, Int64, Single, Double, Decimal |
| SByte | Int16, Int32, Int64, Single, Double, Decimal |
| Int16 | Int32, Int64, Single, Double, Decimal |
| UInt16 | UInt32, Int32, UInt64, Int64, Single, Double, Decimal |
| Char | UInt16, UInt32, Int32, UInt64, Int64, Single, Double, Decimal |
| Int32 | Int64, Double, Decimal |
| UInt32 | Int64, UInt64, Double, Decimal |
| Int64 | Decimal |
| UInt64 | Decimal |
| Single | Double |

Some widening conversions to [Single](#) or [Double](#) can cause a loss of precision. The following table describes the widening conversions that sometimes result in a loss of information.

[] Expand table

| Type | Can be converted to |
|-------|---------------------|
| Int32 | Single |

| Type | Can be converted to |
|---------|---------------------|
| UInt32 | Single |
| Int64 | Single, Double |
| UInt64 | Single, Double |
| Decimal | Single, Double |

Narrowing conversions

A narrowing conversion to [Single](#) or [Double](#) can cause a loss of information. If the target type cannot properly express the magnitude of the source, the resulting type is set to the constant `PositiveInfinity` or `NegativeInfinity`. `PositiveInfinity` results from dividing a positive number by zero and is also returned when the value of a [Single](#) or [Double](#) exceeds the value of the `.MaxValue` field. `NegativeInfinity` results from dividing a negative number by zero and is also returned when the value of a [Single](#) or [Double](#) falls below the value of the `.MinValue` field. A conversion from a [Double](#) to a [Single](#) might result in `PositiveInfinity` or `NegativeInfinity`.

A narrowing conversion can also result in a loss of information for other data types. However, an [OverflowException](#) is thrown if the value of a type that is being converted falls outside of the range specified by the target type's `.MaxValue` and `.MinValue` fields, and the conversion is checked by the runtime to ensure that the value of the target type does not exceed its `.MaxValue` or `.MinValue`. Conversions that are performed with the [System.Convert](#) class are always checked in this manner.

The following table lists conversions that throw an [OverflowException](#) using [System.Convert](#) or any checked conversion if the value of the type being converted is outside the defined range of the resulting type.

[\[+\] Expand table](#)

| Type | Can be converted to |
|--------|------------------------------------|
| Byte | SByte |
| SByte | Byte, UInt16, UInt32, UInt64 |
| Int16 | Byte, SByte, UInt16 |
| UInt16 | Byte, SByte, Int16 |
| Int32 | Byte, SByte, Int16, UInt16, UInt32 |

| Type | Can be converted to |
|---------|--|
| UInt32 | Byte, SByte, Int16, UInt16, Int32 |
| Int64 | Byte, SByte, Int16, UInt16, Int32, UInt32, UInt64 |
| UInt64 | Byte, SByte, Int16, UInt16, Int32, UInt32, Int64 |
| Decimal | Byte, SByte, Int16, UInt16, Int32, UInt32, Int64, UInt64 |
| Single | Byte, SByte, Int16, UInt16, Int32, UInt32, Int64, UInt64 |
| Double | Byte, SByte, Int16, UInt16, Int32, UInt32, Int64, UInt64 |

See also

- [System.Convert](#)
- [Type Conversion in .NET](#)

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

.NET is an open source project.
Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

System.Convert class

Article • 01/08/2024

This article provides supplementary remarks to the reference documentation for this API.

The static [Convert](#) class contains methods that are primarily used to support conversion to and from the base data types in .NET. The supported base types are [Boolean](#), [Char](#), [SByte](#), [Byte](#), [Int16](#), [Int32](#), [Int64](#), [UInt16](#), [UInt32](#), [UInt64](#), [Single](#), [Double](#), [Decimal](#), [DateTime](#) and [String](#). In addition, the [Convert](#) class includes methods to support other kinds of conversions.

Conversions to and from base types

A conversion method exists to convert every base type to every other base type. However, the actual call to a particular conversion method can produce one of five outcomes, depending on the value of the base type at run time and the target base type. These five outcomes are:

- No conversion. This occurs when an attempt is made to convert from a type to itself (for example, by calling [Convert.ToInt32\(Int32\)](#) with an argument of type [Int32](#)). In this case, the method simply returns an instance of the original type.
- An [InvalidOperationException](#). This occurs when a particular conversion is not supported. An [InvalidOperationException](#) is thrown for the following conversions:
 - Conversions from [Char](#) to [Boolean](#), [Single](#), [Double](#), [Decimal](#), or [DateTime](#).
 - Conversions from [Boolean](#), [Single](#), [Double](#), [Decimal](#), or [DateTime](#) to [Char](#).
 - Conversions from [DateTime](#) to any other type except [String](#).
 - Conversions from any other type, except [String](#), to [DateTime](#).
- A [FormatException](#). This occurs when the attempt to convert a string value to any other base type fails because the string is not in the proper format. The exception is thrown for the following conversions:
 - A string to be converted to a [Boolean](#) value does not equal [Boolean.TrueString](#) or [Boolean.FalseString](#).
 - A string to be converted to a [Char](#) value consists of multiple characters.
 - A string to be converted to any numeric type is not recognized as a valid number.
 - A string to be converted to a [DateTime](#) is not recognized as a valid date and time value.

- A successful conversion. For conversions between two different base types not listed in the previous outcomes, all widening conversions as well as all narrowing conversions that do not result in a loss of data will succeed and the method will return a value of the targeted base type.
- An [OverflowException](#). This occurs when a narrowing conversion results in a loss of data. For example, trying to convert a [Int32](#) instance whose value is 10000 to a [Byte](#) type throws an [OverflowException](#) because 10000 is outside the range of the [Byte](#) data type.

An exception will not be thrown if the conversion of a numeric type results in a loss of precision (that is, the loss of some least significant digits). However, an exception will be thrown if the result is larger than can be represented by the particular conversion method's return value type.

For example, when a [Double](#) is converted to a [Single](#), a loss of precision might occur but no exception is thrown. However, if the magnitude of the [Double](#) is too large to be represented by a [Single](#), an overflow exception is thrown.

Non-decimal numbers

The [Convert](#) class includes static methods that you can call to convert integral values to non-decimal string representations, and to convert strings that represent non-decimal numbers to integral values. Each of these conversion methods includes a `base` argument that lets you specify the number system; binary (base 2), octal (base 8), and hexadecimal (base 16), as well as decimal (base 10). There is a set of methods to convert each of the CLS-compliant primitive integral types to a string, and one to convert a string to each of the primitive integral types:

- [ToString\(Byte, Int32\)](#) and [ToByte\(String, Int32\)](#), to convert a byte value to and from a string in a specified base.
- [ToString\(Int16, Int32\)](#) and [ToInt16\(String, Int32\)](#), to convert a 16-bit signed integer to and from a string in a specified base.
- [ToString\(Int32, Int32\)](#) and [ToInt32\(String, Int32\)](#), to convert a 32-bit signed integer to and from a string in a specified base.
- [ToString\(Int64, Int32\)](#) and [ToInt64\(String, Int32\)](#), to convert a 64-bit signed integer to and from a string in a specified base.
- [ToSByte\(String, Int32\)](#), to convert the string representation of a byte value in a specified format to a signed byte.

- [ToInt16\(String, Int32\)](#), to convert the string representation of an integer in a specified format to an unsigned 16-bit integer.
- [ToInt32\(String, Int32\)](#), to convert the string representation of an integer in a specified format to an unsigned 32-bit integer.
- [ToInt64\(String, Int32\)](#), to convert the string representation of an integer in a specified format to an unsigned 64-bit integer.

The following example converts the value of [Int16.MaxValue](#) to a string in all supported numeric formats. It then converts the string value back to a [Int16](#) value.

C#

```
using System;

public class Example
{
    public static void Main()
    {
        int[] baseValues = { 2, 8, 10, 16 };
        short value = Int16.MaxValue;
        foreach (var baseValue in baseValues) {
            String s = Convert.ToString(value, baseValue);
            short value2 = Convert.ToInt16(s, baseValue);

            Console.WriteLine("{0} --> {1} (base {2}) --> {3}",
                value, s, baseValue, value2);
        }
    }
}
// The example displays the following output:
//      32767 --> 11111111111111 (base 2) --> 32767
//      32767 --> 77777 (base 8) --> 32767
//      32767 --> 32767 (base 10) --> 32767
//      32767 --> 7fff (base 16) --> 32767
```

Conversions from custom objects to base types

In addition to supporting conversions between the base types, the [Convert](#) method supports conversion of any custom type to any base type. To do this, the custom type must implement the [IConvertible](#) interface, which defines methods for converting the implementing type to each of the base types. Conversions that are not supported by a particular type should throw an [InvalidOperationException](#).

When the [ChangeType](#) method is passed a custom type as its first parameter, or when the [Convert.ToType](#) method (such as [Convert.ToInt32\(Object\)](#)) or

`Convert.ToDouble(Object, IFormatProvider)` is called and passed an instance of a custom type as its first parameter, the `Convert` method, in turn, calls the custom type's `IConvertible` implementation to perform the conversion. For more information, see [Type Conversion in .NET](#).

Culture-specific formatting information

All the base type conversion methods and the `ChangeType` method include overloads that have a parameter of type `IFormatProvider`. For example, the `Convert.ToBoolean` method has the following two overloads:

- `Convert.ToBoolean(Object, IFormatProvider)`
- `Convert.ToBoolean(String, IFormatProvider)`

The `IFormatProvider` parameter can supply culture-specific formatting information to assist the conversion process. However, it is ignored by most of the base type conversion methods. It is used only by the following base type conversion methods. If a `null` `IFormatProvider` argument is passed to these methods, the `CultureInfo` object that represents the current culture is used.

- By methods that convert a value to a numeric type. The `IFormatProvider` parameter is used by the overload that has parameters of type `String` and `IFormatProvider`. It is also used by the overload that has parameters of type `Object` and `IFormatProvider` if the object's run-time type is a `String`.
- By methods that convert a value to a date and time. The `IFormatProvider` parameter is used by the overload that has parameters of type `String` and `IFormatProvider`. It is also used by the overload that has parameters of type `Object` and `IFormatProvider` if the object's run-time type is a `String`.
- By the `Convert.ToString` overloads that include an `IFormatProvider` parameter and that convert either a numeric value to a string or a `DateTime` value to a string.

However, any user-defined type that implements `IConvertible` can make use of the `IFormatProvider` parameter.

Base64 encoding

Base64 encoding converts binary data to a string. Data expressed as base-64 digits can be easily conveyed over data channels that can only transmit 7-bit characters. The `Convert` class includes the following methods to support base64 encoding: A set of

methods support converting an array of bytes to and from a [String](#) or to and from an array of Unicode characters consisting of base-64 digit characters.

- [ToBase64String](#), which converts a byte array to a base64-encoded string.
- [ToBase64CharArray](#), which converts a byte array to a base64-encoded character array.
- [FromBase64String](#), which converts a base64-encoded string to a byte array.
- [FromBase64CharArray](#), which converts a base64-encoded character array to a byte array.

Other common conversions

You can use other .NET classes to perform some conversions that aren't supported by the static methods of the [Convert](#) class. These include:

- Conversion to byte arrays

The [BitConverter](#) class provides methods that convert the primitive numeric types (including [Boolean](#)) to byte arrays and from byte arrays back to primitive data types.

- Character encoding and decoding

The [Encoding](#) class and its derived classes (such as [UnicodeEncoding](#) and [UTF8Encoding](#)) provide methods to encode a character array or a string (that is, to convert them to a byte array in a particular encoding) and to decode an encoded byte array (that is, convert a byte array back to UTF16-encoded Unicode characters). For more information, see [Character Encoding in .NET](#).

Examples

The following example demonstrates some of the conversion methods in the [Convert](#) class, including [ToInt32](#), [.ToBoolean](#), and [ToString](#).

C#

```
double dNumber = 23.15;

try {
    // Returns 23
    int iNumber = System.Convert.ToInt32(dNumber);
}
catch (System.OverflowException) {
    System.Console.WriteLine(
```

```
        "Overflow in double to int conversion.");
}
// Returns True
bool bNumber = System.Convert.ToBoolean(dNumber);

// Returns "23.15"
string strNumber = System.Convert.ToString(dNumber);

try {
    // Returns '2'
    char chrNumber = System.Convert.ToChar(strNumber[0]);
}
catch (System.ArgumentNullException) {
    System.Console.WriteLine("String is null");
}
catch (System.FormatException) {
    System.Console.WriteLine("String length is greater than 1.");
}

// System.Console.ReadLine() returns a string and it
// must be converted.
int newInteger = 0;
try {
    System.Console.WriteLine("Enter an integer:");
    newInteger = System.Convert.ToInt32(
        System.Console.ReadLine());
}
catch (System.ArgumentNullException) {
    System.Console.WriteLine("String is null.");
}
catch (System.FormatException) {
    System.Console.WriteLine("String does not consist of an " +
        "optional sign followed by a series of digits.");
}
catch (System.OverflowException) {
    System.Console.WriteLine(
        "Overflow in string to int conversion.");
}

System.Console.WriteLine("Your integer as a double is {0}",
    System.Convert.ToDouble(newInteger));
```

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For



.NET feedback

.NET is an open source project.
Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

more information, see [our contributor guide](#).

Choosing between anonymous and tuple types

Article • 03/09/2023

Choosing the appropriate type involves considering its usability, performance, and tradeoffs compared to other types. Anonymous types have been available since C# 3.0, while generic [System.Tuple<T1,T2>](#) types were introduced with .NET Framework 4.0. Since then new options have been introduced with language level support, such as [System.ValueTuple<T1,T2>](#) - which as the name implies, provide a value type with the flexibility of anonymous types. In this article, you'll learn when it's appropriate to choose one type over the other.

Usability and functionality

Anonymous types were introduced in C# 3.0 with Language-Integrated Query (LINQ) expressions. With LINQ, developers often project results from queries into anonymous types that hold a few select properties from the objects they're working with. Consider the following example, that instantiates an array of [DateTime](#) objects, and iterates through them projecting into an anonymous type with two properties.

C#

```
var dates = new[]
{
    DateTime.UtcNow.AddHours(-1),
    DateTime.UtcNow,
    DateTime.UtcNow.AddHours(1),
};

foreach (var anonymous in
    dates.Select(
        date => new { Formatted = $"{date:MMM dd, yyyy hh:mm zzz}", date.Ticks }))
{
    Console.WriteLine($"Ticks: {anonymous.Ticks}, Formatted: {anonymous.Formatted}");
}
```

Anonymous types are instantiated by using the [new](#) operator, and the property names and types are inferred from the declaration. If two or more anonymous object initializers in the same assembly specify a sequence of properties that are in the same order and

that have the same names and types, the compiler treats the objects as instances of the same type. They share the same compiler-generated type information.

The previous C# snippet projects an anonymous type with two properties, much like the following compiler-generated C# class:

```
C#  
  
internal sealed class f__AnonymousType0  
{  
    public string Formatted { get; }  
    public long Ticks { get; }  
  
    public f__AnonymousType0(string formatted, long ticks)  
    {  
        Formatted = formatted;  
        Ticks = ticks;  
    }  
}
```

For more information, see [anonymous types](#). The same functionality exists with tuples when projecting into LINQ queries, you can select properties into tuples. These tuples flow through the query, just as anonymous types would. Now consider the following example using the `System.Tuple<string, long>`.

```
C#  
  
var dates = new[]  
{  
    DateTime.UtcNow.AddHours(-1),  
    DateTime.UtcNow,  
    DateTime.UtcNow.AddHours(1),  
};  
  
foreach (var tuple in  
    dates.Select(  
        date => new Tuple<string, long>(${date:MMM dd, yyyy hh:mm  
zzz}), date.Ticks)))  
{  
    Console.WriteLine($"Ticks: {tuple.Item2}, formatted: {tuple.Item1}");  
}
```

With the `System.Tuple<T1,T2>`, the instance exposes numbered item properties, such as `Item1`, and `Item2`. These property names can make it more difficult to understand the intent of the property values, as the property name only provides the ordinal. Furthermore, the `System.Tuple` types are reference `class` types. The `System.ValueTuple<T1,T2>` however, is a value `struct` type. The following C# snippet,

uses `ValueTuple<string, long>` to project into. In doing so, it assigns using a literal syntax.

```
C#  
  
var dates = new[]  
{  
    DateTime.UtcNow.AddHours(-1),  
    DateTime.UtcNow,  
    DateTime.UtcNow.AddHours(1),  
};  
  
foreach (var (formatted, ticks) in  
    dates.Select(  
        date => (Formatted: ${date:MMM dd, yyyy at hh:mm zzz},  
date.Ticks)))  
{  
    Console.WriteLine($"Ticks: {ticks}, formatted: {formatted}");  
}
```

For more information about tuples, see [Tuple types \(C# reference\)](#) or [Tuples \(Visual Basic\)](#).

The previous examples are all functionally equivalent, however, there are slight differences in their usability and their underlying implementations.

Tradeoffs

You might want to always use `ValueTuple` over `Tuple`, and anonymous types, but there are tradeoffs you should consider. The `ValueTuple` types are mutable, whereas `Tuple` are read-only. Anonymous types can be used in expression trees, while tuples cannot. The following table is an overview of some of the key differences.

Key differences

| Name | Access modifier | Type | Custom member name | Deconstruction support | Expression tree support |
|-------------------------|-----------------------|---------------------|--------------------|------------------------|-------------------------|
| Anonymous types | <code>internal</code> | <code>class</code> | ✓ | ✗ | ✓ |
| <code>Tuple</code> | <code>public</code> | <code>class</code> | ✗ | ✗ | ✓ |
| <code>ValueTuple</code> | <code>public</code> | <code>struct</code> | ✓ | ✓ | ✗ |

Serialization

One important consideration when choosing a type, is whether or not it will need to be serialized. Serialization is the process of converting the state of an object into a form that can be persisted or transported. For more information, see [serialization](#). When serialization is important, creating a `class` or `struct` is preferred over anonymous types or tuple types.

Performance

Performance between these types depends on the scenario. The major impact involves the tradeoff between allocations and copying. In most scenarios, the impact is small. When major impacts could arise, measurements should be taken to inform the decision.

Conclusion

As a developer choosing between tuples and anonymous types, there are several factors to consider. Generally speaking, if you're not working with [expression trees](#), and you're comfortable with tuple syntax then choose [ValueTuple](#) as they provide a value type with the flexibility to name properties. If you're working with expression trees, and you'd prefer to name properties, choose anonymous types. Otherwise, use [Tuple](#).

See also

- [Anonymous types](#)
- [Expression trees](#)
- [Tuple types \(C# reference\)](#)
- [Tuples \(Visual Basic\)](#)
- [Type design guidelines](#)

 Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

more information, see [our contributor guide](#).

.NET class library overview

Article • 04/19/2023

.NET APIs include classes, interfaces, delegates, and value types that expedite and optimize the development process and provide access to system functionality. To facilitate interoperability between languages, most .NET types are CLS-compliant and can therefore be used from any programming language whose compiler conforms to the common language specification (CLS).

.NET types are the foundation on which .NET applications, components, and controls are built. .NET includes types that perform the following functions:

- Represent base data types and exceptions.
- Encapsulate data structures.
- Perform I/O.
- Access information about loaded types.
- Invoke .NET security checks.
- Provide data access, rich client-side GUI, and server-controlled, client-side GUI.

.NET provides a rich set of interfaces, as well as abstract and concrete (non-abstract) classes. You can use the concrete classes as-is or, in many cases, derive your own classes from them. To use the functionality of an interface, you can either create a class that implements the interface or derive a class from one of the .NET classes that implements the interface.

Naming conventions

.NET types use a dot syntax naming scheme that connotes a hierarchy. This technique groups related types into namespaces so they can be searched and referenced more easily. The first part of the full name—up to the rightmost dot—is the namespace name. The last part of the name is the type name. For example,

`System.Collections.Generic.List<T>` represents the `List<T>` type, which belongs to the `System.Collections.Generic` namespace. The types in `System.Collections.Generic` can be used to work with generic collections.

This naming scheme makes it easy for library developers extending .NET to create hierarchical groups of types and name them in a consistent, informative manner. It also allows types to be unambiguously identified by their full name (that is, by their namespace and type name), which prevents type name collisions. Library developers are expected to use the following convention when creating names for their namespaces:

CompanyName.TechologyName

For example, the namespace `Microsoft.Word` conforms to this guideline.

The use of naming patterns to group related types into namespaces is a useful way to build and document class libraries. However, this naming scheme has no effect on visibility, member access, inheritance, security, or binding. A namespace can be partitioned across multiple assemblies and a single assembly can contain types from multiple namespaces. The assembly provides the formal structure for versioning, deployment, security, loading, and visibility in the common language runtime.

For more information on namespaces and type names, see [Common Type System](#).

System namespace

The `System` namespace is the root namespace for fundamental types in .NET. This namespace includes classes that represent the base data types used by all applications, for example, `Object` (the root of the inheritance hierarchy), `Byte`, `Char`, `Array`, `Int32`, and `String`. Many of these types correspond to the primitive data types that your programming language uses. When you write code using .NET types, you can use your language's corresponding keyword when a .NET base data type is expected.

The following table lists the base types that .NET supplies, briefly describes each type, and indicates the corresponding type in Visual Basic, C#, C++, and F#.

| Category | Class name | Description | Visual Basic data type | C# data type | C++/CLI data type | F# data type |
|----------|--------------------|--|------------------------|--------------------|----------------------------------|--------------------|
| Integer | <code>Byte</code> | An 8-bit unsigned integer. | <code>Byte</code> | <code>byte</code> | <code>unsigned char</code> | <code>byte</code> |
| | <code>SByte</code> | An 8-bit signed integer. Not CLS-compliant. | <code>SByte</code> | <code>sbyte</code> | <code>char or signed char</code> | <code>sbyte</code> |
| | <code>Int16</code> | A 16-bit signed integer. | <code>Short</code> | <code>short</code> | <code>short</code> | <code>int16</code> |
| | <code>Int32</code> | A 32-bit signed integer. | <code>Integer</code> | <code>int</code> | <code>int or long</code> | <code>int</code> |
| | <code>Int64</code> | A 64-bit signed integer. | <code>Long</code> | <code>long</code> | <code>_int64</code> | <code>int64</code> |

| Category | Class name | Description | Visual Basic data type | C# data type | C++/CLI data type | F# data type |
|----------------|------------|---|------------------------|--------------|-------------------------------|-------------------|
| | UInt16 | A 16-bit unsigned integer. Not CLS-compliant. | UShort | ushort | unsigned short | uint16 |
| | UInt32 | A 32-bit unsigned integer. Not CLS-compliant. | UInteger | uint | unsigned int or unsigned long | uint32 |
| | UInt64 | A 64-bit unsigned integer. Not CLS-compliant. | ULong | ulong | unsigned __int64 | uint64 |
| Floating point | Half | A half-precision (16-bit) floating-point number. | | | | |
| | Single | A single-precision (32-bit) floating-point number. | Single | float | float | float32 or single |
| | Double | A double-precision (64-bit) floating-point number. | Double | double | double | float or double |
| Logical | Boolean | A Boolean value (true or false). | Boolean | bool | bool | bool |
| Other | Char | A Unicode (16-bit) character. | Char | char | wchar_t | char |
| | Decimal | A decimal (128-bit) value. | Decimal | decimal | Decimal | decimal |
| | IntPtr | A signed integer whose size depends on the underlying platform (a 32-bit value on a 32-bit platform and a 64-bit value on a 64-bit platform). | | nint | | unativeint |
| | UIntPtr | An unsigned integer whose size depends | | nuint | | unativeint |

| Category | Class name | Description | Visual Basic data type | C# data type | C++/CLI data type | F# data type |
|----------|------------|---|---------------------------|--------------|-------------------|--------------|
| | | on the underlying platform (a 32-bit value on a 32-bit platform and a 64-bit value on a 64-bit platform). | | | | |
| | | Not CLS-compliant. | | | | |
| | Object | The root of the object hierarchy. | Object | object | Object^ | obj |
| | String | An immutable, fixed-length string of Unicode characters. | String | string | String^ | string |

In addition to the base data types, the [System](#) namespace contains over 100 classes, ranging from classes that handle exceptions to classes that deal with core runtime concepts, such as application domains and the garbage collector. The [System](#) namespace also contains many second-level namespaces.

For more information about namespaces, use the [.NET API Browser](#) to browse the .NET Class Library. The API reference documentation provides documentation on each namespace, its types, and each of their members.

Data structures

.NET includes a set of data structures that are the workhorses of many .NET apps. These are mostly collections, but also include other types.

- [Array](#) - Represents an array of strongly typed objects that can be accessed by index. Has a fixed size, per its construction.
- [List<T>](#) - Represents a strongly typed list of objects that can be accessed by index. Is automatically resized as needed.
- [Dictionary< TKey, TValue >](#) - Represents a collection of values that are indexed by a key. Values can be accessed via key. Is automatically resized as needed.
- [Uri](#) - Provides an object representation of a uniform resource identifier (URI) and easy access to the parts of the URI.
- [DateTime](#) - Represents an instant in time, typically expressed as a date and time of day.

Utility APIs

.NET includes a set of utility APIs that provide functionality for many important tasks.

- [HttpClient](#) - An API for sending HTTP requests and receiving HTTP responses from a resource identified by a URI.
- [XDocument](#) - An API for loading and querying XML documents with LINQ.
- [StreamReader](#) - An API for reading files.
- [StreamWriter](#) - An API for writing files.

App-model APIs

There are many app models that can be used with .NET, for example:

- [ASP.NET](#) - A web framework for building web sites and services. Supported on Windows, Linux, and macOS (depends on ASP.NET version).
- [.NET MAUI](#) - An app platform for building native apps that run on Windows, macOS, iOS, and Android using C#.
- [Windows Desktop](#) - Includes Windows Presentation Foundation (WPF) and Windows Forms.

See also

- [Runtime libraries overview](#)
- [Common type system](#)
- [.NET API browser](#)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

System.Object class

Article • 01/08/2024

This article provides supplementary remarks to the reference documentation for this API.

The [Object](#) class is the ultimate base class of all .NET classes; it is the root of the type hierarchy.

Because all classes in .NET are derived from [Object](#), every method defined in the [Object](#) class is available in all objects in the system. Derived classes can and do override some of these methods, including:

- [Equals](#): Supports comparisons between objects.
- [Finalize](#): Performs cleanup operations before an object is automatically reclaimed.
- [GetHashCode](#): Generates a number corresponding to the value of the object to support the use of a hash table.
- [ToString](#): Manufactures a human-readable text string that describes an instance of the class.

Languages typically don't require a class to declare inheritance from [Object](#) because the inheritance is implicit.

Performance considerations

If you're designing a class, such as a collection, that must handle any type of object, you can create class members that accept instances of the [Object](#) class. However, the process of boxing and unboxing a type carries a performance cost. If you know your new class will frequently handle certain value types you can use one of two tactics to minimize the cost of boxing.

- Create a general method that accepts an [Object](#) type, and a set of type-specific method overloads that accept each value type you expect your class to frequently handle. If a type-specific method exists that accepts the calling parameter type, no boxing occurs and the type-specific method is invoked. If there is no method argument that matches the calling parameter type, the parameter is boxed and the general method is invoked.
- Design your type and its members to use [generics](#). The common language runtime creates a closed generic type when you create an instance of your class and specify a generic type argument. The generic method is type-specific and can be invoked without boxing the calling parameter.

Although it's sometimes necessary to develop general purpose classes that accept and return [Object](#) types, you can improve performance by also providing a type-specific class to handle a frequently used type. For example, providing a class that is specific to setting and getting Boolean values eliminates the cost of boxing and unboxing Boolean values.

 [Collaborate with us on GitHub](#)

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

 .NET

.NET feedback

.NET is an open source project.
Select a link to provide feedback:

-  [Open a documentation issue](#)
-  [Provide product feedback](#)

System.Nullable class

Article • 01/08/2024

This article provides supplementary remarks to the reference documentation for this API.

The [Nullable](#) class supports value types that can be assigned `null`.

A type is said to be nullable if it can be assigned a value or can be assigned `null`, which means the type has no value whatsoever. By default, all reference types, such as [String](#), are nullable, but all value types, such as [Int32](#), are not.

In C# and Visual Basic, you mark a value type as nullable by using the `?` notation after the value type. For example, `int?` in C# or `Integer?` in Visual Basic declares an integer value type that can be assigned `null`.

The [Nullable](#) class provides complementary support for the [Nullable<T>](#) structure. The [Nullable](#) class supports obtaining the underlying type of a nullable type, and comparison and equality operations on pairs of nullable types whose underlying value type does not support generic comparison and equality operations.

Boxing and unboxing

When a nullable type is boxed, the common language runtime automatically boxes the underlying value of the [Nullable<T>](#) object, not the [Nullable<T>](#) object itself. That is, if the [HasValue](#) property is `true`, the contents of the [Value](#) property is boxed.

If the [HasValue](#) property of a nullable type is `false`, the result of the boxing operation is `null`. When the underlying value of a nullable type is unboxed, the common language runtime creates a new [Nullable<T>](#) structure initialized to the underlying value.

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For

.NET

.NET feedback

.NET is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

more information, see [our contributor guide](#).

Generics in .NET

Article • 02/17/2023

Generics let you tailor a method, class, structure, or interface to the precise data type it acts upon. For example, instead of using the [Hashtable](#) class, which allows keys and values to be of any type, you can use the [Dictionary<TKey,TValue>](#) generic class and specify the types allowed for the key and the value. Among the benefits of generics are increased code reusability and type safety.

Define and use generics

Generics are classes, structures, interfaces, and methods that have placeholders (type parameters) for one or more of the types that they store or use. A generic collection class might use a type parameter as a placeholder for the type of objects that it stores. The type parameters appear as the types of its fields and the parameter types of its methods. A generic method might use its type parameter as the type of its return value or as the type of one of its formal parameters.

The following code illustrates a simple generic class definition.

```
C#  
  
public class SimpleGenericClass<T>  
{  
    public T Field;  
}
```

When you create an instance of a generic class, you specify the actual types to substitute for the type parameters. This establishes a new generic class, referred to as a constructed generic class, with your chosen types substituted everywhere that the type parameters appear. The result is a type-safe class that is tailored to your choice of types, as the following code illustrates.

```
C#  
  
public static void Main()  
{  
    SimpleGenericClass<string> g = new SimpleGenericClass<string>();  
    g.Field = "A string";  
    //...  
    Console.WriteLine("SimpleGenericClass.Field           = \"{0}\\"",  
        g.Field);  
    Console.WriteLine("SimpleGenericClass.Field.GetType() = {0}",
```

```
g.Field.GetType().FullName);  
}
```

Terminology

The following terms are used to discuss generics in .NET:

- A *generic type definition* is a class, structure, or interface declaration that functions as a template, with placeholders for the types that it can contain or use. For example, the [System.Collections.Generic.Dictionary<TKey,TValue>](#) class can contain two types: keys and values. Because a generic type definition is only a template, you cannot create instances of a class, structure, or interface that is a generic type definition.
- *Generic type parameters*, or *type parameters*, are the placeholders in a generic type or method definition. The [System.Collections.Generic.Dictionary<TKey,TValue>](#) generic type has two type parameters, `TKey` and `TValue`, that represent the types of its keys and values.
- A *constructed generic type*, or *constructed type*, is the result of specifying types for the generic type parameters of a generic type definition.
- A *generic type argument* is any type that is substituted for a generic type parameter.
- The general term *generic type* includes both constructed types and generic type definitions.
- *Covariance* and *contravariance* of generic type parameters enable you to use constructed generic types whose type arguments are more derived (covariance) or less derived (contravariance) than a target constructed type. Covariance and contravariance are collectively referred to as *variance*. For more information, see [Covariance and contravariance](#).
- *Constraints* are limits placed on generic type parameters. For example, you might limit a type parameter to types that implement the [System.Collections.Generic.IComparer<T>](#) generic interface, to ensure that instances of the type can be ordered. You can also constrain type parameters to types that have a particular base class, that have a parameterless constructor, or that are reference types or value types. Users of the generic type cannot substitute type arguments that do not satisfy the constraints.

- A *generic method definition* is a method with two parameter lists: a list of generic type parameters and a list of formal parameters. Type parameters can appear as the return type or as the types of the formal parameters, as the following code shows.

```
C#  
  
T MyGenericMethod<T>(T arg)  
{  
    T temp = arg;  
    //...  
    return temp;  
}
```

Generic methods can appear on generic or nongeneric types. It's important to note that a method is not generic just because it belongs to a generic type, or even because it has formal parameters whose types are the generic parameters of the enclosing type. A method is generic only if it has its own list of type parameters. In the following code, only method `G` is generic.

```
C#  
  
class A  
{  
    T G<T>(T arg)  
    {  
        T temp = arg;  
        //...  
        return temp;  
    }  
}  
class MyGenericClass<T>  
{  
    T M(T arg)  
    {  
        T temp = arg;  
        //...  
        return temp;  
    }  
}
```

Advantages and disadvantages of generics

There are many advantages to using generic collections and delegates:

- Type safety. Generics shift the burden of type safety from you to the compiler. There is no need to write code to test for the correct data type because it is

enforced at compile time. The need for type casting and the possibility of run-time errors are reduced.

- Less code and code is more easily reused. There is no need to inherit from a base type and override members. For example, the [LinkedList<T>](#) is ready for immediate use. For example, you can create a linked list of strings with the following variable declaration:

C#

```
LinkedList<string> llist = new LinkedList<string>();
```

- Better performance. Generic collection types generally perform better for storing and manipulating value types because there is no need to box the value types.
- Generic delegates enable type-safe callbacks without the need to create multiple delegate classes. For example, the [Predicate<T>](#) generic delegate allows you to create a method that implements your own search criteria for a particular type and to use your method with methods of the [Array](#) type such as [Find](#), [FindLast](#), and [FindAll](#).
- Generics streamline dynamically generated code. When you use generics with dynamically generated code you do not need to generate the type. This increases the number of scenarios in which you can use lightweight dynamic methods instead of generating entire assemblies. For more information, see [How to: Define and Execute Dynamic Methods](#) and [DynamicMethod](#).

The following are some limitations of generics:

- Generic types can be derived from most base classes, such as [MarshalByRefObject](#) (and constraints can be used to require that generic type parameters derive from base classes like [MarshalByRefObject](#)). However, .NET does not support context-bound generic types. A generic type can be derived from [ContextBoundObject](#), but trying to create an instance of that type causes a [TypeLoadException](#).
- Enumerations cannot have generic type parameters. An enumeration can be generic only incidentally (for example, because it is nested in a generic type that is defined using Visual Basic, C#, or C++). For more information, see "Enumerations" in [Common Type System](#).
- Lightweight dynamic methods cannot be generic.
- In Visual Basic, C#, and C++, a nested type that is enclosed in a generic type cannot be instantiated unless types have been assigned to the type parameters of

all enclosing types. Another way of saying this is that in reflection, a nested type that is defined using these languages includes the type parameters of all its enclosing types. This allows the type parameters of enclosing types to be used in the member definitions of a nested type. For more information, see "Nested Types" in [MakeGenericType](#).

 **Note**

A nested type that is defined by emitting code in a dynamic assembly or by using the [Ilasm.exe \(IL Assembler\)](#) is not required to include the type parameters of its enclosing types; however, if it does not include them, the type parameters are not in scope in the nested class.

For more information, see "Nested Types" in [MakeGenericType](#).

Class library and language support

.NET provides a number of generic collection classes in the following namespaces:

- The [System.Collections.Generic](#) namespace contains most of the generic collection types provided by .NET, such as the [List<T>](#) and [Dictionary< TKey,TValue >](#) generic classes.
- The [System.Collections.ObjectModel](#) namespace contains additional generic collection types, such as the [ReadOnlyCollection<T>](#) generic class, that are useful for exposing object models to users of your classes.

Generic interfaces for implementing sort and equality comparisons are provided in the [System](#) namespace, along with generic delegate types for event handlers, conversions, and search predicates.

The [System.Numerics](#) namespace provides generic interfaces for mathematical functionality (available in .NET 7 and later versions). For more information, see [Generic math](#).

Support for generics has been added to the [System.Reflection](#) namespace for examining generic types and generic methods, to [System.Reflection.Emit](#) for emitting dynamic assemblies that contain generic types and methods, and to [System.CodeDom](#) for generating source graphs that include generics.

The common language runtime provides new opcodes and prefixes to support generic types in Microsoft intermediate language (MSIL), including [Stelem](#), [Ldelem](#), [Unbox_Any](#),

Constrained, and [Readonly](#).

Visual C++, C#, and Visual Basic all provide full support for defining and using generics. For more information about language support, see [Generic Types in Visual Basic](#), [Introduction to Generics](#), and [Overview of Generics in Visual C++](#).

Nested types and generics

A type that is nested in a generic type can depend on the type parameters of the enclosing generic type. The common language runtime considers nested types to be generic, even if they do not have generic type parameters of their own. When you create an instance of a nested type, you must specify type arguments for all enclosing generic types.

Related articles

| Title | Description |
|---|--|
| Generic Collections in .NET | Describes generic collection classes and other generic types in .NET. |
| Generic Delegates for Manipulating Arrays and Lists | Describes generic delegates for conversions, search predicates, and actions to be taken on elements of an array or collection. |
| Generic math | Describes how you can perform mathematical operations generically. |
| Generic Interfaces | Describes generic interfaces that provide common functionality across families of generic types. |
| Covariance and Contravariance | Describes covariance and contravariance in generic type parameters. |
| Commonly Used Collection Types | Provides summary information about the characteristics and usage scenarios of the collection types in .NET, including generic types. |
| When to Use Generic Collections | Describes general rules for determining when to use generic collection types. |
| How to: Define a Generic Type with Reflection Emit | Explains how to generate dynamic assemblies that include generic types and methods. |
| Generic Types in Visual Basic | Describes the generics feature for Visual Basic users, including how-to topics for using and defining generic types. |

| Title | Description |
|--|---|
| Introduction to Generics | Provides an overview of defining and using generic types for C# users. |
| Overview of Generics in Visual C++ | Describes the generics feature for C++ users, including the differences between generics and templates. |

Reference

- [System.Collections.Generic](#)
- [System.Collections.ObjectModel](#)
- [System.Reflection.Emit.OpCodes](#)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

Generic types overview

Article • 07/23/2022

Developers use generics all the time in .NET, whether implicitly or explicitly. When you use LINQ in .NET, did you ever notice that you're working with `IEnumerable<T>`? Or if you ever saw an online sample of a "generic repository" for talking to databases using Entity Framework, did you see that most methods return `IQueryable<T>`? You may have wondered what the `T` is in these examples and why it's in there.

First introduced in .NET Framework 2.0, generics are essentially a "code template" that allows developers to define `type-safe` data structures without committing to an actual data type. For example, `List<T>` is a `generic collection` that can be declared and used with any type, such as `List<int>`, `List<string>`, or `List<Person>`.

To understand why generics are useful, let's take a look at a specific class before and after adding generics: `ArrayList`. In .NET Framework 1.0, the `ArrayList` elements were of type `Object`. Any element added to the collection was silently converted into an `Object`. The same would happen when reading elements from the list. This process is known as `boxing` and `unboxing`, and it impacts performance. Aside from performance, however, there's no way to determine the type of data in the list at compile time, which makes for some fragile code. Generics solve this problem by defining the type of data each instance of list will contain. For example, you can only add integers to `List<int>` and only add Persons to `List<Person>`.

Generics are also available at run time. The runtime knows what type of data structure you're using and can store it in memory more efficiently.

The following example is a small program that illustrates the efficiency of knowing the data structure type at run time:

C#

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Diagnostics;

namespace GenericsExample {
    class Program {
        static void Main(string[] args) {
            //generic list
            List<int> ListGeneric = new List<int> { 5, 9, 1, 4 };
            //non-generic list
            ArrayList ListNonGeneric = new ArrayList { 5, 9, 1, 4 };
        }
    }
}
```

```

    // timer for generic list sort
    Stopwatch s = Stopwatch.StartNew();
    ListGeneric.Sort();
    s.Stop();
    Console.WriteLine($"Generic Sort: {ListGeneric} \n Time taken:
{s.Elapsed.TotalMilliseconds}ms");

    //timer for non-generic list sort
    Stopwatch s2 = Stopwatch.StartNew();
    ListNonGeneric.Sort();
    s2.Stop();
    Console.WriteLine($"Non-Generic Sort: {ListNonGeneric} \n Time
taken: {s2.Elapsed.TotalMilliseconds}ms");
    Console.ReadLine();
}
}
}

```

This program produces output similar to the following:

Console

```

Generic Sort: System.Collections.Generic.List`1[System.Int32]
Time taken: 0.0034ms
Non-Generic Sort: System.Collections.ArrayList
Time taken: 0.2592ms

```

The first thing you can notice here is that sorting the generic list is significantly faster than sorting the non-generic list. You might also notice that the type for the generic list is distinct ([System.Int32]), whereas the type for the non-generic list is generalized. Because the runtime knows the generic `List<int>` is of type `Int32`, it can store the list elements in an underlying integer array in memory, while the non-generic `ArrayList` has to cast each list element to an object. As this example shows, the extra casts take up time and slow down the list sort.

An additional advantage of the runtime knowing the type of your generic is a better debugging experience. When you're debugging a generic in C#, you know what type each element is in your data structure. Without generics, you would have no idea what type each element was.

See also

- [C# Programming Guide - Generics](#)

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

Generic collections in .NET

Article • 09/15/2021

The .NET class library provides a number of generic collection classes in the [System.Collections.Generic](#) and [System.Collections.ObjectModel](#) namespaces. For more detailed information about these classes, see [Commonly Used Collection Types](#).

System.Collections.Generic

Many of the generic collection types are direct analogs of nongeneric types.

[Dictionary<TKey,TValue>](#) is a generic version of [Hashtable](#); it uses the generic structure [KeyValuePair<TKey,TValue>](#) for enumeration instead of [DictionaryEntry](#).

[List<T>](#) is a generic version of [ArrayList](#). There are generic [Queue<T>](#) and [Stack<T>](#) classes that correspond to the nongeneric versions.

There are generic and nongeneric versions of [SortedList<TKey,TValue>](#). Both versions are hybrids of a dictionary and a list. The [SortedDictionary<TKey,TValue>](#) generic class is a pure dictionary and has no nongeneric counterpart.

The [LinkedList<T>](#) generic class is a true linked list. It has no nongeneric counterpart.

System.Collections.ObjectModel

The [Collection<T>](#) generic class provides a base class for deriving your own generic collection types. The [ReadOnlyCollection<T>](#) class provides an easy way to produce a read-only collection from any type that implements the [IList<T>](#) generic interface. The [KeyedCollection<TKey,TItem>](#) generic class provides a way to store objects that contain their own keys.

Other generic types

The [Nullable<T>](#) generic structure allows you to use value types as if they could be assigned `null`. This can be useful when working with database queries, where fields that contain value types can be missing. The generic type parameter can be any value type.

Note

In C# and Visual Basic, it is not necessary to use [Nullable<T>](#) explicitly because the language has syntax for nullable types. See [Nullable value types \(C# reference\)](#)

and Nullable value types (Visual Basic).

The `ArraySegment<T>` generic structure provides a way to delimit a range of elements within a one-dimensional, zero-based array of any type. The generic type parameter is the type of the array's elements.

The `EventHandler<TEventArgs>` generic delegate eliminates the need to declare a delegate type to handle events, if your event follows the event-handling pattern used by .NET. For example, suppose you have created a `MyEventArgs` class, derived from `EventArgs`, to hold the data for your event. You can then declare the event as follows:

C#

```
public event EventHandler<MyEventArgs> MyEvent;
```

See also

- [System.Collections.Generic](#)
- [System.Collections.ObjectModel](#)
- [Generics](#)
- [Generic Delegates for Manipulating Arrays and Lists](#)
- [Generic Interfaces](#)

 Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

Generic Delegates for Manipulating Arrays and Lists

Article • 09/15/2021

This topic provides an overview of generic delegates for conversions, search predicates, and actions to be taken on elements of an array or collection.

Generic Delegates for Manipulating Arrays and Lists

The [Action<T>](#) generic delegate represents a method that performs some action on an element of the specified type. You can create a method that performs the desired action on the element, create an instance of the [Action<T>](#) delegate to represent that method, and then pass the array and the delegate to the [Array.ForEach](#) static generic method. The method is called for every element of the array.

The [List<T>](#) generic class also provides a [ForEach](#) method that uses the [Action<T>](#) delegate. This method is not generic.

ⓘ Note

This makes an interesting point about generic types and methods. The [Array.ForEach](#) method must be static ([Shared](#) in Visual Basic) and generic because [Array](#) is not a generic type; the only reason you can specify a type for [Array.ForEach](#) to operate on is that the method has its own type parameter list. By contrast, the nongeneric [List<T>.ForEach](#) method belongs to the generic class [List<T>](#), so it simply uses the type parameter of its class. The class is strongly typed, so the method can be an instance method.

The [Predicate<T>](#) generic delegate represents a method that determines whether a particular element meets criteria you define. You can use it with the following static generic methods of [Array](#) to search for an element or a set of elements: [Exists](#), [Find](#), [FindAll](#), [FindIndex](#), [FindLast](#), [FindLastIndex](#), and [TrueForAll](#).

[Predicate<T>](#) also works with the corresponding nongeneric instance methods of the [List<T>](#) generic class.

The [Comparison<T>](#) generic delegate allows you to provide a sort order for array or list elements that do not have a native sort order, or to override the native sort order.

Create a method that performs the comparison, create an instance of the [Comparison<T>](#) delegate to represent your method, and then pass the array and the delegate to the [Array.Sort<T>\(T\[\], Comparison<T>\)](#) static generic method. The [List<T>](#) generic class provides a corresponding instance method overload, [List<T>.Sort\(Comparison<T>\)](#).

The [Converter<TInput,TOutput>](#) generic delegate allows you to define a conversion between two types, and to convert an array of one type into an array of the other, or to convert a list of one type to a list of the other. Create a method that converts the elements of the existing list to a new type, create a delegate instance to represent the method, and use the [Array.ConvertAll](#) generic static method to produce an array of the new type from the original array, or the [List<T>.ConvertAll<TOutput>\(Converter<T,TOutput>\)](#) generic instance method to produce a list of the new type from the original list.

Chaining Delegates

Many of the methods that use these delegates return an array or list, which can be passed to another method. For example, if you want to select certain elements of an array, convert those elements to a new type, and save them in a new array, you can pass the array returned by the [FindAll](#) generic method to the [ConvertAll](#) generic method. If the new element type lacks a natural sort order, you can pass the array returned by the [ConvertAll](#) generic method to the [Sort<T>\(T\[\], Comparison<T>\)](#) generic method.

See also

- [System.Collections.Generic](#)
- [System.Collections.ObjectModel](#)
- [Generics](#)
- [Generic Collections in the .NET](#)
- [Generic Interfaces](#)
- [Covariance and Contravariance](#)

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

more information, see [our contributor guide](#).

 [Provide product feedback](#)

Generic math

Article • 04/21/2023

.NET 7 introduces new math-related generic interfaces to the base class library. The availability of these interfaces means you can constrain a type parameter of a generic type or method to be "number-like". In addition, C# 11 and later lets you define [static virtual interface members](#). Because operators must be declared as `static`, this new C# feature lets operators be declared in the new interfaces for number-like types.

Together, these innovations allow you to perform mathematical operations generically—that is, without having to know the exact type you're working with. For example, if you wanted to write a method that adds two numbers, previously you had to add an overload of the method for each type (for example, `static int Add(int first, int second)` and `static float Add(float first, float second)`). Now you can write a single, generic method, where the type parameter is constrained to be a number-like type. For example:

C#

```
static T Add<T>(T left, T right)
    where T : INumber<TSelf>
{
    return left + right;
}
```

In this method, the type parameter `T` is constrained to be a type that implements the new [INumber<TSelf>](#) interface. [INumber<TSelf>](#) implements the [IAdditionOperators<TSelf,TOther,TResult>](#) interface, which contains the `+ operator`. That allows the method to generically add the two numbers. The method can be used with any of .NET's built-in numeric types, because they've all been updated to implement [INumber<TSelf>](#) in .NET 7.

Library authors will benefit most from the generic math interfaces, because they can simplify their code base by removing "redundant" overloads. Other developers will benefit indirectly, because the APIs they consume may start supporting more types.

The interfaces

The interfaces were designed to be both fine-grained enough that users can define their own interfaces on top, while also being granular enough that they're easy to consume. To that extent, there are a few core numeric interfaces that most users will interact with,

such as [INumber<TSelf>](#) and [IBinaryInteger<TSelf>](#). The more fine-grained interfaces, such as [IAdditionOperators<TSelf,TOther,TResult>](#) and [ITrigonometricFunctions<TSelf>](#), support these types and are available for developers who define their own domain-specific numeric interfaces.

- [Numeric interfaces](#)
- [Operator interfaces](#)
- [Function interfaces](#)
- [Parsing and formatting interfaces](#)

Numeric interfaces

This section describes the interfaces in [System.Numerics](#) that describe number-like types and the functionality available to them.

| Interface name | Description |
|--|--|
| IBinaryFloatingPointIEEE754<TSelf> | Exposes APIs common to <i>binary</i> floating-point types ¹ that implement the IEEE 754 standard. |
| IBinaryInteger<TSelf> | Exposes APIs common to binary integers ² . |
| IBinaryNumber<TSelf> | Exposes APIs common to binary numbers. |
| IFloatingPoint<TSelf> | Exposes APIs common to floating-point types. |
| IFloatingPointIEEE754<TSelf> | Exposes APIs common to floating-point types that implement the IEEE 754 standard. |
| INumber<TSelf> | Exposes APIs common to comparable number types (effectively the "real" number domain). |
| INumberBase<TSelf> | Exposes APIs common to all number types (effectively the "complex" number domain). |
| ISignedNumber<TSelf> | Exposes APIs common to all signed number types (such as the concept of <code>NegativeOne</code>). |
| IUnsignedNumber<TSelf> | Exposes APIs common to all unsigned number types. |
| IAdditiveIdentity<TSelf,TResult> | Exposes the concept of <code>(x + T.AdditiveIdentity) == x</code> . |
| IMaxValue<TSelf> | Exposes the concept of <code>T.MinValue</code> and <code>T.MaxValue</code> . |
| IMultiplicativeIdentity<TSelf,TResult> | Exposes the concept of <code>(x * T.MultiplicativeIdentity) == x</code> . |

¹The *binary* floating-point types are `Double` (`double`), `Half`, and `Single` (`float`).

²The binary integer types are `Byte` (`byte`), `Int16` (`short`), `Int32` (`int`), `Int64` (`long`), `Int128`, `IntPtr` (`nint`), `SByte` (`sbyte`), `UInt16` (`ushort`), `UInt32` (`uint`), `UInt64` (`ulong`), `UInt128`, and `UIntPtr` (`nuint`).

The interface you're most likely to use directly is `INumber<TSelf>`, which roughly corresponds to a *real* number. If a type implements this interface, it means that a value has a sign (this includes `unsigned` types, which are considered positive) and can be compared to other values of the same type. `INumberBase<TSelf>` confers more advanced concepts, such as *complex* and *imaginary* numbers, for example, the square root of a negative number. Other interfaces, such as `IFloatingPointeee754<TSelf>`, were created because not all operations make sense for all number types—for example, calculating the floor of a number only makes sense for floating-point types. In the .NET base class library, the floating-point type `Double` implements `IFloatingPointeee754<TSelf>` but `Int32` doesn't.

Several of the interfaces are also implemented by various other types, including `Char`, `DateOnly`, `DateTime`, `DateTimeOffset`, `Decimal`, `Guid`, `TimeOnly`, and `TimeSpan`.

The following table shows some of the core APIs exposed by each interface.

| Interface | API name | Description |
|--|--------------------------------|---|
| <code>IBinaryInteger<TSelf></code> | <code>DivRem</code> | Computes the quotient and remainder simultaneously. |
| | <code>LeadingZeroCount</code> | Counts the number of leading zero bits in the binary representation. |
| | <code>PopCount</code> | Counts the number of set bits in the binary representation. |
| | <code>RotateLeft</code> | Rotates bits left, sometimes also called a circular left shift. |
| | <code>RotateRight</code> | Rotates bits right, sometimes also called a circular right shift. |
| | <code>TrailingZeroCount</code> | Counts the number of trailing zero bits in the binary representation. |
| <code>IFloatingPoint<TSelf></code> | <code>Ceiling</code> | Rounds the value towards positive infinity. +4.5 becomes +5, and -4.5 becomes -4. |
| | <code>Floor</code> | Rounds the value towards negative infinity. +4.5 becomes +4, and -4.5 becomes -5. |
| | <code>Round</code> | Rounds the value using the specified |

| Interface | API name | Description |
|-----------------------|------------------|---|
| | Round | rounding mode. |
| | Truncate | Rounds the value towards zero. +4.5 becomes +4, and -4.5 becomes -4. |
| IFloatingPoint<TSelf> | E | Gets a value representing Euler's number for the type. |
| | Epsilon | Gets the smallest representable value that's greater than zero for the type. |
| | NaN | Gets a value representing <code>NaN</code> for the type. |
| | NegativeInfinity | Gets a value representing <code>-Infinity</code> for the type. |
| | NegativeZero | Gets a value representing <code>-Zero</code> for the type. |
| | Pi | Gets a value representing <code>Pi</code> for the type. |
| | PositiveInfinity | Gets a value representing <code>+Infinity</code> for the type. |
| | Tau | Gets a value representing <code>Tau</code> ($2 * \Pi$) for the type. |
| | (Other) | (Implements the full set of interfaces listed under Function interfaces .) |
| INumber<TSelf> | Clamp | Restricts a value to no more and no less than the specified min and max value. |
| | CopySign | Sets the sign of a specified value to the same as another specified value. |
| | Max | Returns the greater of two values, returning <code>NaN</code> if either input is <code>NaN</code> . |
| | MaxNumber | Returns the greater of two values, returning the number if one input is <code>NaN</code> . |
| | Min | Returns the lesser of two values, returning <code>NaN</code> if either input is <code>NaN</code> . |
| | MinNumber | Returns the lesser of two values, returning the number if one input is <code>NaN</code> . |
| | Sign | Returns -1 for negative values, 0 for zero, and +1 for positive values. |

| Interface | API name | Description |
|---------------------------------------|--------------------------------|---|
| <code>INumberBase<TSelf></code> | <code>One</code> | Gets the value 1 for the type. |
| | <code>Radix</code> | Gets the radix, or base, for the type. <code>Int32</code> returns 2. <code>Decimal</code> returns 10. |
| | <code>Zero</code> | Gets the value 0 for the type. |
| | <code>CreateChecked</code> | Creates a value, throwing an <code>OverflowException</code> if the input can't fit. ¹ |
| | <code>CreateSaturating</code> | Creates a value, clamping to <code>T.MinValue</code> or <code>T.MaxValue</code> if the input can't fit. ¹ |
| | <code>CreateTruncating</code> | Creates a value from another value, wrapping around if the input can't fit. ¹ |
| | <code>IsComplexNumber</code> | Returns true if the value has a non-zero real part and a non-zero imaginary part. |
| | <code>IsEvenInteger</code> | Returns true if the value is an even integer. 2.0 returns <code>true</code> , and 2.2 returns <code>false</code> . |
| | <code>IsFinite</code> | Returns true if the value is not infinite and not <code>NaN</code> . |
| | <code>IsImaginaryNumber</code> | Returns true if the value has a zero real part. This means <code>0</code> is imaginary and <code>1 + 1i</code> isn't. |
| | <code>IsInfinity</code> | Returns true if the value represents infinity. |
| | <code>IsInteger</code> | Returns true if the value is an integer. 2.0 and 3.0 return <code>true</code> , and 2.2 and 3.1 return <code>false</code> . |
| | <code>IsNaN</code> | Returns true if the value represents <code>NaN</code> . |
| | <code>IsNegative</code> | Returns true if the value is negative. This includes -0.0. |
| | <code>IsPositive</code> | Returns true if the value is positive. This includes 0 and +0.0. |
| | <code>IsRealNumber</code> | Returns true if the value has a zero imaginary part. This means 0 is real as are all <code>INumber<T></code> types. |
| | <code>IsZero</code> | Returns true if the value represents zero. This includes 0, +0.0, and -0.0. |

| Interface | API name | Description |
|---|--------------------------|---|
| | MaxMagnitude | Returns the value with a greater absolute value, returning <code>NaN</code> if either input is <code>NaN</code> . |
| | MaxMagnitudeNumber | Returns the value with a greater absolute value, returning the number if one input is <code>NaN</code> . |
| | MinMagnitude | Returns the value with a lesser absolute value, returning <code>NaN</code> if either input is <code>NaN</code> . |
| | MinMagnitudeNumber | Returns the value with a lesser absolute value, returning the number if one input is <code>NaN</code> . |
| <code>ISignedNumber<TSelf></code> | <code>NegativeOne</code> | Gets the value -1 for the type. |

¹To help understand the behavior of the three `Create*` methods, consider the following examples.

Example when given a value that's too large:

- `byte.CreateChecked(384)` will throw an [OverflowException](#).
- `byte.CreateSaturating(384)` returns 255 because 384 is greater than `Byte.MaxValue` (which is 255).
- `byte.CreateTruncating(384)` returns 128 because it takes the lowest 8 bits (384 has a hex representation of `0x0180`, and the lowest 8 bits is `0x80`, which is 128).

Example when given a value that's too small:

- `byte.CreateChecked(-384)` will throw an [OverflowException](#).
- `byte.CreateSaturating(-384)` returns 0 because -384 is smaller than `Byte.MinValue` (which is 0).
- `byte.CreateTruncating(-384)` returns 128 because it takes the lowest 8 bits (384 has a hex representation of `0xFE80`, and the lowest 8 bits is `0x80`, which is 128).

The `Create*` methods also have some special considerations for IEEE 754 floating-point types, like `float` and `double`, as they have the special values `PositiveInfinity`, `NegativeInfinity`, and `NaN`. All three `Create*` APIs behave as `CreateSaturating`. Also, while `.MinValue` and `.MaxValue` represent the largest negative/positive "normal" number, the actual minimum and maximum values are `NegativeInfinity` and `PositiveInfinity`, so they clamp to these values instead.

Operator interfaces

The operator interfaces correspond to the various operators available to the C# language.

- They explicitly don't pair operations such as multiplication and division since that isn't correct for all types. For example, `Matrix4x4 * Matrix4x4` is valid, but `Matrix4x4 / Matrix4x4` isn't valid.
- They typically allow the input and result types to differ to support scenarios such as dividing two integers to obtain a `double`, for example, `3 / 2 = 1.5`, or calculating the average of a set of integers.

| Interface name | Defined operators |
|---|---|
| IAdditionOperators<TSelf,TOther,TResult> | <code>x + y</code> |
| IBitwiseOperators<TSelf,TOther,TResult> | <code>x & y</code> , <code>x y</code> , <code>x ^ y</code> , and <code>~x</code> |
| IComparisonOperators<TSelf,TOther,TResult> | <code>x < y</code> , <code>x > y</code> , <code>x <= y</code> , and <code>x >= y</code> |
| IDecrementOperators<TSelf> | <code>--x</code> and <code>x--</code> |
| IDivisionOperators<TSelf,TOther,TResult> | <code>x / y</code> |
| IEqualityOperators<TSelf,TOther,TResult> | <code>x == y</code> and <code>x != y</code> |
| IIncrementOperators<TSelf> | <code>++x</code> and <code>x++</code> |
| IModulusOperators<TSelf,TOther,TResult> | <code>x % y</code> |
| IMultiplyOperators<TSelf,TOther,TResult> | <code>x * y</code> |
| IShiftOperators<TSelf,TOther,TResult> | <code>x << y</code> and <code>x >> y</code> |
| ISubtractionOperators<TSelf,TOther,TResult> | <code>x - y</code> |
| IUnaryNegationOperators<TSelf,TResult> | <code>-x</code> |
| IUnaryPlusOperators<TSelf,TResult> | <code>+x</code> |

ⓘ Note

Some of the interfaces define a checked operator in addition to a regular unchecked operator. Checked operators are called in checked contexts and allow a user-defined type to define overflow behavior. If you implement a checked

operator, for example, `CheckedSubtraction(TSelf, TOther)`, you must also implement the unchecked operator, for example, `Subtraction(TSelf, TOther)`.

Function interfaces

The function interfaces define common mathematical APIs that apply more broadly than to a specific [numeric interface](#). These interfaces are all implemented by `IFloatingPoint<TSelf>`, and may get implemented by other relevant types in the future.

| Interface name | Description |
|---|--|
| <code>IExponentialFunctions<TSelf></code> | Exposes exponential functions supporting <code>e^x</code> , <code>e^x - 1</code> , <code>2^x</code> , <code>2^x - 1</code> , <code>10^x</code> , and <code>10^x - 1</code> . |
| <code>IHyperbolicFunctions<TSelf></code> | Exposes hyperbolic functions supporting <code>acosh(x)</code> , <code>asinh(x)</code> , <code>atanh(x)</code> , <code>cosh(x)</code> , <code>sinh(x)</code> , and <code>tanh(x)</code> . |
| <code>ILogarithmicFunctions<TSelf></code> | Exposes logarithmic functions supporting <code>ln(x)</code> , <code>ln(x + 1)</code> , <code>log2(x)</code> , <code>log2(x + 1)</code> , <code>log10(x)</code> , and <code>log10(x + 1)</code> . |
| <code>IPowerFunctions<TSelf></code> | Exposes power functions supporting <code>x^y</code> . |
| <code>IRootFunctions<TSelf></code> | Exposes root functions supporting <code>cbrt(x)</code> and <code>sqrt(x)</code> . |
| <code>ITrigonometricFunctions<TSelf></code> | Exposes trigonometric functions supporting <code>acos(x)</code> , <code>asin(x)</code> , <code>atan(x)</code> , <code>cos(x)</code> , <code>sin(x)</code> , and <code>tan(x)</code> . |

Parsing and formatting interfaces

Parsing and formatting are core concepts in programming. They're commonly used when converting user input to a given type or displaying a type to the user. These interfaces are in the [System](#) namespace.

| Interface name | Description |
|--|---|
| <code>IParsable<TSelf></code> | Exposes support for <code>T.Parse(string, IFormatProvider)</code> and <code>T.TryParse(string, IFormatProvider, out TSelf)</code> . |
| <code>ISpanParsable<TSelf></code> | Exposes support for <code>T.Parse(ReadOnlySpan<char>, IFormatProvider)</code> and <code>T.TryParse(ReadOnlySpan<char>, IFormatProvider, out TSelf)</code> . |
| <code>IFormattable</code> ¹ | Exposes support for <code>value.ToString(string, IFormatProvider)</code> . |
| <code>ISpanFormattable</code> ¹ | Exposes support for <code>value.TryFormat(Span<char>, out int,</code> |

| Interface name | Description |
|----------------|--|
| | <code>ReadOnlySpan<char>, IFormatProvider).</code> |

¹This interface isn't new, nor is it generic. However, it's implemented by all number types and represents the inverse operation of `IParsable`.

For example, the following program takes two numbers as input, reading them from the console using a generic method where the type parameter is constrained to be `IParsable<TSelf>`. It calculates the average using a generic method where the type parameters for the input and result values are constrained to be `INumber<TSelf>`, and then displays the result to the console.

C#

```
using System.Globalization;
using System.Numerics;

static TResult Average<T, TResult>(T first, T second)
    where T : INumber<T>
    where TResult : INumber<TResult>
{
    return TResult.CreateChecked( (first + second) / T.CreateChecked(2) );
}

static T ParseInvariant<T>(string s)
    where T : IParsable<T>
{
    return T.Parse(s, CultureInfo.InvariantCulture);
}

Console.Write("First number: ");
var left = ParseInvariant<float>(Console.ReadLine());

Console.Write("Second number: ");
var right = ParseInvariant<float>(Console.ReadLine());

Console.WriteLine($"Result: {Average<float, float>(left, right)}");

/* This code displays output similar to:

First number: 5.0
Second number: 6
Result: 5.5
*/
```

See also

- [Generic math in .NET 7 \(blog post\)](#) ↗

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

Generic interfaces in .NET

Article • 08/03/2022

This article provides an overview of .NET's generic interfaces that provide common functionality across families of generic types.

Generic interfaces provide type-safe counterparts to nongeneric interfaces for ordering and equality comparisons, and for functionality that's shared by generic collection types. .NET 7 introduces generic interfaces for number-like types, for example, [System.Numerics.INumber<TSelf>](#). These interfaces let you define generic methods that provide mathematical functionality, where the generic type parameter is constrained to be a type that implements a generic, numeric interface.

ⓘ Note

The type parameters of several generic interfaces are marked covariant or contravariant, providing greater flexibility in assigning and using types that implement these interfaces. For more information, see [Covariance and Contravariance](#).

Equality and ordering comparisons

- In the [System](#) namespace, the [System.IComparable<T>](#) and [System.IEquatable<T>](#) generic interfaces, like their nongeneric counterparts, define methods for ordering comparisons and equality comparisons, respectively. Types implement these interfaces to provide the ability to perform such comparisons.
- In the [System.Collections.Generic](#) namespace, the [IComparer<T>](#) and [IEqualityComparer<T>](#) generic interfaces offer a way to define an ordering or equality comparison for types that don't implement the [System.IComparable<T>](#) or [System.IEquatable<T>](#) interface. They also provide a way to redefine those relationships for types that do.

These interfaces are used by methods and constructors of many of the generic collection classes. For example, you can pass a generic [IComparer<T>](#) object to the constructor of the [SortedDictionary<TKey,TValue>](#) class to specify a sort order for a type that does not implement generic [System.IComparable<T>](#). There are overloads of the [Array.Sort](#) generic static method and the [List<T>.Sort](#) instance method for sorting arrays and lists using generic [IComparer<T>](#) implementations.

The [Comparer<T>](#) and [EqualityComparer<T>](#) generic classes provide base classes for implementations of the [IComparer<T>](#) and [IEqualityComparer<T>](#) generic interfaces, and also provide default ordering and equality comparisons through their respective [Comparer<T>.Default](#) and [EqualityComparer<T>.Default](#) properties.

Collection functionality

- The [ICollection<T>](#) generic interface is the basic interface for generic collection types. It provides basic functionality for adding, removing, copying, and enumerating elements. [ICollection<T>](#) inherits from both generic [IEnumerable<T>](#) and nongeneric [IEnumerable](#).
- The [IList<T>](#) generic interface extends the [ICollection<T>](#) generic interface with methods for indexed retrieval.
- The [IDictionary< TKey, TValue >](#) generic interface extends the [ICollection<T>](#) generic interface with methods for keyed retrieval. Generic dictionary types in the .NET base class library also implement the nongeneric [IDictionary](#) interface.
- The [IEnumerable<T>](#) generic interface provides a generic enumerator structure. The [IEnumerator<T>](#) generic interface implemented by generic enumerators inherits the nongeneric [IEnumerator](#) interface; the [MoveNext](#) and [Reset](#) members, which do not depend on the type parameter [T](#), appear only on the nongeneric interface. This means that any consumer of the nongeneric interface can also consume the generic interface.

Mathematical functionality

.NET 7 introduces generic interfaces in the [System.Numerics](#) namespace that describe number-like types and the functionality available to them. The 20 numeric types that the .NET base class library provides, for example, [Int32](#) and [Double](#), have been updated to implement these interfaces. The most prominent of these interfaces is [INumber< TSelf >](#), which roughly corresponds to a "real" number.

For more information about these interfaces, see [Generic math](#).

See also

- [System.Collections.Generic](#)
- [System.Collections.ObjectModel](#)

- Generics
- Generic collections in .NET
- Generic delegates for manipulating arrays and lists
- Covariance and contravariance

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

-  [Open a documentation issue](#)
-  [Provide product feedback](#)

Covariance and contravariance in generics

Article • 09/15/2021

Covariance and *contravariance* are terms that refer to the ability to use a more derived type (more specific) or a less derived type (less specific) than originally specified. Generic type parameters support covariance and contravariance to provide greater flexibility in assigning and using generic types.

When you're referring to a type system, covariance, contravariance, and invariance have the following definitions. The examples assume a base class named `Base` and a derived class named `Derived`.

- **Covariance**

Enables you to use a more derived type than originally specified.

You can assign an instance of `IEnumerable<Derived>` to a variable of type `IEnumerable<Base>`.

- **Contravariance**

Enables you to use a more generic (less derived) type than originally specified.

You can assign an instance of `Action<Base>` to a variable of type `Action<Derived>`.

- **Invariance**

Means that you can use only the type originally specified. An invariant generic type parameter is neither covariant nor contravariant.

You cannot assign an instance of `List<Base>` to a variable of type `List<Derived>` or vice versa.

Covariant type parameters enable you to make assignments that look much like ordinary [Polymorphism](#), as shown in the following code.

```
C#
```

```
IEnumerable<Derived> d = new List<Derived>();  
IEnumerable<Base> b = d;
```

The `List<T>` class implements the `IEnumerable<T>` interface, so `List<Derived>` (`List(Of Derived)` in Visual Basic) implements `IEnumerable<Derived>`. The covariant type parameter does the rest.

Contravariance, on the other hand, seems counterintuitive. The following example creates a delegate of type `Action<Base>` (`Action(Of Base)` in Visual Basic), and then assigns that delegate to a variable of type `Action<Derived>`.

C#

```
Action<Base> b = (target) => { Console.WriteLine(target.GetType().Name); };
Action<Derived> d = b;
d(new Derived());
```

This seems backward, but it is type-safe code that compiles and runs. The lambda expression matches the delegate it's assigned to, so it defines a method that takes one parameter of type `Base` and that has no return value. The resulting delegate can be assigned to a variable of type `Action<Derived>` because the type parameter `T` of the `Action<T>` delegate is contravariant. The code is type-safe because `T` specifies a parameter type. When the delegate of type `Action<Base>` is invoked as if it were a delegate of type `Action<Derived>`, its argument must be of type `Derived`. This argument can always be passed safely to the underlying method, because the method's parameter is of type `Base`.

In general, a covariant type parameter can be used as the return type of a delegate, and contravariant type parameters can be used as parameter types. For an interface, covariant type parameters can be used as the return types of the interface's methods, and contravariant type parameters can be used as the parameter types of the interface's methods.

Covariance and contravariance are collectively referred to as *variance*. A generic type parameter that is not marked covariant or contravariant is referred to as *invariant*. A brief summary of facts about variance in the common language runtime:

- Variant type parameters are restricted to generic interface and generic delegate types.
- A generic interface or generic delegate type can have both covariant and contravariant type parameters.
- Variance applies only to reference types; if you specify a value type for a variant type parameter, that type parameter is invariant for the resulting constructed type.

- Variance does not apply to delegate combination. That is, given two delegates of types `Action<Derived>` and `Action<Base>` (`Action(Of Derived)` and `Action(Of Base)` in Visual Basic), you cannot combine the second delegate with the first although the result would be type safe. Variance allows the second delegate to be assigned to a variable of type `Action<Derived>`, but delegates can combine only if their types match exactly.
- Starting in C# 9, covariant return types are supported. An overriding method can declare a more derived return type the method it overrides, and an overriding, read-only property can declare a more derived type.

Generic interfaces with covariant type parameters

Several generic interfaces have covariant type parameters, for example, `IEnumerable<T>`, `IEnumerator<T>`, `IQueryable<T>`, and `IGrouping< TKey, TElement >`. All the type parameters of these interfaces are covariant, so the type parameters are used only for the return types of the members.

The following example illustrates covariant type parameters. The example defines two types: `Base` has a static method named `PrintBases` that takes an `IEnumerable<Base>` (`IEnumerable(Of Base)` in Visual Basic) and prints the elements. `Derived` inherits from `Base`. The example creates an empty `List<Derived>` (`List(Of Derived)` in Visual Basic) and demonstrates that this type can be passed to `PrintBases` and assigned to a variable of type `IEnumerable<Base>` without casting. `List<T>` implements `IEnumerable<T>`, which has a single covariant type parameter. The covariant type parameter is the reason why an instance of `IEnumerable<Derived>` can be used instead of `IEnumerable<Base>`.

C#

```
using System;
using System.Collections.Generic;

class Base
{
    public static void PrintBases(IEnumerable<Base> bases)
    {
        foreach(Base b in bases)
        {
            Console.WriteLine(b);
        }
    }
}
```

```

class Derived : Base
{
    public static void Main()
    {
        List<Derived> dlist = new List<Derived>();

        Derived.PrintBases(dlist);
        IEnumerable<Base> bIEnum = dlist;
    }
}

```

Generic interfaces with contravariant type parameters

Several generic interfaces have contravariant type parameters; for example:

`IComparer<T>`, `IComparable<T>`, and `IEqualityComparer<T>`. These interfaces have only contravariant type parameters, so the type parameters are used only as parameter types in the members of the interfaces.

The following example illustrates contravariant type parameters. The example defines an abstract (`MustInherit` in Visual Basic) `Shape` class with an `Area` property. The example also defines a `ShapeAreaComparer` class that implements `IComparer<Shape>` (`IComparer(Of Shape)` in Visual Basic). The implementation of the `IComparer<T>.Compare` method is based on the value of the `Area` property, so `ShapeAreaComparer` can be used to sort `Shape` objects by area.

The `Circle` class inherits `Shape` and overrides `Area`. The example creates a `SortedSet<T>` of `circle` objects, using a constructor that takes an `IComparer<Circle>` (`IComparer(Of Circle)` in Visual Basic). However, instead of passing an `IComparer<Circle>`, the example passes a `ShapeAreaComparer` object, which implements `IComparer<Shape>`. The example can pass a comparer of a less derived type (`Shape`) when the code calls for a comparer of a more derived type (`Circle`), because the type parameter of the `IComparer<T>` generic interface is contravariant.

When a new `Circle` object is added to the `SortedSet<Circle>`, the `IComparer<Shape>.Compare` method (`IComparer(Of Shape).Compare` method in Visual Basic) of the `ShapeAreaComparer` object is called each time the new element is compared to an existing element. The parameter type of the method (`Shape`) is less derived than the type that is being passed (`Circle`), so the call is type safe. Contravariance enables `ShapeAreaComparer` to sort a collection of any single type, as well as a mixed collection of types, that derive from `Shape`.

C#

```
using System;
using System.Collections.Generic;

abstract class Shape
{
    public virtual double Area { get { return 0; } }
}

class Circle : Shape
{
    private double r;
    public Circle(double radius) { r = radius; }
    public double Radius { get { return r; } }
    public override double Area { get { return Math.PI * r * r; } }
}

class ShapeAreaComparer : System.Collections.Generic.IComparer<Shape>
{
    int IComparer<Shape>.Compare(Shape a, Shape b)
    {
        if (a == null) return b == null ? 0 : -1;
        return b == null ? 1 : a.Area.CompareTo(b.Area);
    }
}

class Program
{
    static void Main()
    {
        // You can pass ShapeAreaComparer, which implements
        IComparer<Shape>,
        // even though the constructor for SortedSet<Circle> expects
        // IComparer<Circle>, because type parameter T of IComparer<T> is
        // contravariant.
        SortedSet<Circle> circlesByArea =
            new SortedSet<Circle>(new ShapeAreaComparer())
            { new Circle(7.2), new Circle(100), null, new Circle(.01) };

        foreach (Circle c in circlesByArea)
        {
            Console.WriteLine(c == null ? "null" : "Circle with area " +
c.Area);
        }
    }
}

/* This code example produces the following output:

null
Circle with area 0.000314159265358979
Circle with area 162.860163162095
```

```
Circle with area 31415.9265358979
*/
```

Generic delegates with variant type parameters

The `Func` generic delegates, such as `Func<T,TResult>`, have covariant return types and contravariant parameter types. The `Action` generic delegates, such as `Action<T1,T2>`, have contravariant parameter types. This means that the delegates can be assigned to variables that have more derived parameter types and (in the case of the `Func` generic delegates) less derived return types.

ⓘ Note

The last generic type parameter of the `Func` generic delegates specifies the type of the return value in the delegate signature. It is covariant (`out` keyword), whereas the other generic type parameters are contravariant (`in` keyword).

The following code illustrates this. The first piece of code defines a class named `Base`, a class named `Derived` that inherits `Base`, and another class with a `static` method (`Shared` in Visual Basic) named `MyMethod`. The method takes an instance of `Base` and returns an instance of `Derived`. (If the argument is an instance of `Derived`, `MyMethod` returns it; if the argument is an instance of `Base`, `MyMethod` returns a new instance of `Derived`.) In `Main()`, the example creates an instance of `Func<Base, Derived>` (`Func(Of Base, Derived)` in Visual Basic) that represents `MyMethod`, and stores it in the variable `f1`.

C#

```
public class Base {}
public class Derived : Base {}

public class Program
{
    public static Derived MyMethod(Base b)
    {
        return b as Derived ?? new Derived();
    }

    static void Main()
    {
        Func<Base, Derived> f1 = MyMethod;
```

The second piece of code shows that the delegate can be assigned to a variable of type `Func<Base, Base>` (`Func(Of Base, Base)` in Visual Basic), because the return type is covariant.

C#

```
// Covariant return type.  
Func<Base, Base> f2 = f1;  
Base b2 = f2(new Base());
```

The third piece of code shows that the delegate can be assigned to a variable of type `Func<Derived, Derived>` (`Func(Of Derived, Derived)` in Visual Basic), because the parameter type is contravariant.

C#

```
// Contravariant parameter type.  
Func<Derived, Derived> f3 = f1;  
Derived d3 = f3(new Derived());
```

The final piece of code shows that the delegate can be assigned to a variable of type `Func<Derived, Base>` (`Func(Of Derived, Base)` in Visual Basic), combining the effects of the contravariant parameter type and the covariant return type.

C#

```
// Covariant return type and contravariant parameter type.  
Func<Derived, Base> f4 = f1;  
Base b4 = f4(new Derived());
```

Variance in non-generic delegates

In the preceding code, the signature of `MyMethod` exactly matches the signature of the constructed generic delegate: `Func<Base, Derived>` (`Func(Of Base, Derived)` in Visual Basic). The example shows that this generic delegate can be stored in variables or method parameters that have more derived parameter types and less derived return types, as long as all the delegate types are constructed from the generic delegate type `Func<T,TResult>`.

This is an important point. The effects of covariance and contravariance in the type parameters of generic delegates are similar to the effects of covariance and contravariance in ordinary delegate binding (see [Variance in Delegates \(C#\)](#) and [Variance in Delegates \(Visual Basic\)](#)). However, variance in delegate binding works with all

delegate types, not just with generic delegate types that have variant type parameters. Furthermore, variance in delegate binding enables a method to be bound to any delegate that has more restrictive parameter types and a less restrictive return type, whereas the assignment of generic delegates works only if both delegate types are constructed from the same generic type definition.

The following example shows the combined effects of variance in delegate binding and variance in generic type parameters. The example defines a type hierarchy that includes three types, from least derived (`Type1`) to most derived (`Type3`). Variance in ordinary delegate binding is used to bind a method with a parameter type of `Type1` and a return type of `Type3` to a generic delegate with a parameter type of `Type2` and a return type of `Type2`. The resulting generic delegate is then assigned to another variable whose generic delegate type has a parameter of type `Type3` and a return type of `Type1`, using the covariance and contravariance of generic type parameters. The second assignment requires both the variable type and the delegate type to be constructed from the same generic type definition, in this case, `Func<T,TResult>`.

C#

```
using System;

public class Type1 {}
public class Type2 : Type1 {}
public class Type3 : Type2 {}

public class Program
{
    public static Type3 MyMethod(Type1 t)
    {
        return t as Type3 ?? new Type3();
    }

    static void Main()
    {
        Func<Type2, Type2> f1 = MyMethod;

        // Covariant return type and contravariant parameter type.
        Func<Type3, Type1> f2 = f1;
        Type1 t1 = f2(new Type3());
    }
}
```

Define variant generic interfaces and delegates

Visual Basic and C# have keywords that enable you to mark the generic type parameters of interfaces and delegates as covariant or contravariant.

A covariant type parameter is marked with the `out` keyword (`Out` keyword in Visual Basic). You can use a covariant type parameter as the return value of a method that belongs to an interface, or as the return type of a delegate. You cannot use a covariant type parameter as a generic type constraint for interface methods.

 **Note**

If a method of an interface has a parameter that is a generic delegate type, a covariant type parameter of the interface type can be used to specify a contravariant type parameter of the delegate type.

A contravariant type parameter is marked with the `in` keyword (`In` keyword in Visual Basic). You can use a contravariant type parameter as the type of a parameter of a method that belongs to an interface, or as the type of a parameter of a delegate. You can use a contravariant type parameter as a generic type constraint for an interface method.

Only interface types and delegate types can have variant type parameters. An interface or delegate type can have both covariant and contravariant type parameters.

Visual Basic and C# do not allow you to violate the rules for using covariant and contravariant type parameters, or to add covariance and contravariance annotations to the type parameters of types other than interfaces and delegates.

For information and example code, see [Variance in Generic Interfaces \(C#\)](#) and [Variance in Generic Interfaces \(Visual Basic\)](#).

List of types

The following interface and delegate types have covariant and/or contravariant type parameters.

| Type | Covariant type parameters | Contravariant type parameters |
|--|---------------------------|-------------------------------|
| <code>Action<T></code> to <code>Action<T1,T2,T3,T4,T5,T6,T7,T8,T9,T10,T11,T12,T13,T14,T15,T16></code> | | Yes |
| <code>Comparison<T></code> | | Yes |
| <code>Converter<TInput, TOutput></code> | Yes | Yes |
| <code>Func<TResult></code> | Yes | |

| Type | Covariant Yes type parameters | Contravariant Yes type parameters |
|--|--|--|
| Func<T,TResult> to Func<T1,T2,T3,T4,T5,T6,T7,T8,T9,T10,T11,T12,T13,T14,T15,T16,TResult> | | |
| IComparable<T> | | Yes |
| Predicate<T> | | Yes |
| IComparer<T> | | Yes |
| IEnumerable<T> | Yes | |
| IEnumerator<T> | Yes | |
| IEqualityComparer<T> | | Yes |
| IGrouping< TKey, TElement > | Yes | |
| IOrderedEnumerable< TElement > | Yes | |
| IOrderedQueryable<T> | Yes | |
| IQueryable<T> | Yes | |

See also

- [Covariance and Contravariance \(C#\)](#)
- [Covariance and Contravariance \(Visual Basic\)](#)
- [Variance in Delegates \(C#\)](#)
- [Variance in Delegates \(Visual Basic\)](#)

 Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

Collections and Data Structures

Article • 08/12/2022

Similar data can often be handled more efficiently when stored and manipulated as a collection. You can use the [System.Array](#) class or the classes in the [System.Collections](#), [System.Collections.Generic](#), [System.Collections.Concurrent](#), and [System.Collections.Immutable](#) namespaces to add, remove, and modify either individual elements or a range of elements in a collection.

There are two main types of collections; generic collections and non-generic collections. Generic collections are type-safe at compile time. Because of this, generic collections typically offer better performance. Generic collections accept a type parameter when they're constructed. They don't require that you cast to and from the [Object](#) type when you add or remove items from the collection. In addition, most generic collections are supported in Windows Store apps. Non-generic collections store items as [Object](#), require casting, and most aren't supported for Windows Store app development. However, you might see non-generic collections in older code.

In .NET Framework 4 and later versions, the collections in the [System.Collections.Concurrent](#) namespace provide efficient thread-safe operations for accessing collection items from multiple threads. The immutable collection classes in the [System.Collections.Immutable](#) namespace ([NuGet package](#)) are inherently thread-safe because operations are performed on a copy of the original collection, and the original collection can't be modified.

Common collection features

All collections provide methods for adding, removing, or finding items in the collection. In addition, all collections that directly or indirectly implement the [ICollection](#) interface or the [ICollection<T>](#) interface share these features:

- **The ability to enumerate the collection**

.NET collections either implement [System.Collections.IEnumerable](#) or [System.Collections.Generic.IEnumerable<T>](#) to enable the collection to be iterated through. An enumerator can be thought of as a movable pointer to any element in the collection. The `foreach`, `in` statement and the [For Each...Next Statement](#) use the enumerator exposed by the [GetEnumerator](#) method and hide the complexity of manipulating the enumerator. In addition, any collection that implements [System.Collections.Generic.IEnumerable<T>](#) is considered a *queryable type* and can be queried with LINQ. LINQ queries provide a common pattern for accessing data. They're typically more concise and readable than standard `foreach` loops and provide filtering, ordering, and grouping capabilities. LINQ queries can also improve performance. For more information, see [LINQ to Objects \(C#\)](#), [LINQ to Objects \(Visual Basic\)](#), [Parallel LINQ \(PLINQ\)](#), [Introduction to LINQ Queries \(C#\)](#), and [Basic Query Operations \(Visual Basic\)](#).

- **The ability to copy the collection contents to an array**

All collections can be copied to an array using the `CopyTo` method. However, the order of the elements in the new array is based on the sequence in which the enumerator returns them. The resulting array is always one-dimensional with a lower bound of zero.

In addition, many collection classes contain the following features:

- **Capacity and Count properties**

The capacity of a collection is the number of elements it can contain. The count of a collection is the number of elements it actually contains. Some collections hide the capacity or the count or both.

Most collections automatically expand in capacity when the current capacity is reached. The memory is reallocated, and the elements are copied from the old collection to the new one. This design reduces the code required to use the collection. However, the performance of the collection might be negatively affected. For example, for `List<T>`, if `Count` is less than `Capacity`, adding an item is an $O(1)$ operation. If the capacity needs to be increased to accommodate the new element, adding an item becomes an $O(n)$ operation, where n is `Count`. The best way to avoid poor performance caused by multiple reallocations is to set the initial capacity to be the estimated size of the collection.

A `BitArray` is a special case; its capacity is the same as its length, which is the same as its count.

- **A consistent lower bound**

The lower bound of a collection is the index of its first element. All indexed collections in the `System.Collections` namespaces have a lower bound of zero, meaning they're 0-indexed. `Array` has a lower bound of zero by default, but a different lower bound can be defined when creating an instance of the `Array` class using `Array.CreateInstance`.

- **Synchronization for access from multiple threads** (`System.Collections` classes only).

Non-generic collection types in the `System.Collections` namespace provide some thread safety with synchronization; typically exposed through the `SyncRoot` and `IsSynchronized` members. These collections aren't thread-safe by default. If you require scalable and efficient multi-threaded access to a collection, use one of the classes in the `System.Collections.Concurrent` namespace or consider using an immutable collection. For more information, see [Thread-Safe Collections](#).

Choose a collection

In general, you should use generic collections. The following table describes some common collection scenarios and the collection classes you can use for those scenarios. If you're new to generic collections, the following table will help you choose the generic collection that works best for your task:

| I want to... | Generic collection options | Non-generic collection options | Thread-safe or immutable collection options |
|---|--|--|--|
| Store items as key/value pairs for quick look-up by key | Dictionary<TKey,TValue> | Hashtable (A collection of key/value pairs that are organized based on the hash code of the key.) | ConcurrentDictionary<TKey,TValue> ReadOnlyDictionary<TKey,TValue> ImmutableDictionary<TKey,TValue> |
| Access items by index | List<T> | Array ArrayList | ImmutableList<T> ImmutableArray |
| Use items first-in-first-out (FIFO) | Queue<T> | Queue | ConcurrentQueue<T> ImmutableQueue<T> |
| Use data Last-In-First-Out (LIFO) | Stack<T> | Stack | ConcurrentStack<T> ImmutableStack<T> |
| Access items sequentially | LinkedList<T> | No recommendation | No recommendation |
| Receive notifications when items are removed or added to the collection. (implements INotifyPropertyChanged and INotifyCollectionChanged) | ObservableCollection<T> | No recommendation | No recommendation |
| A sorted collection | SortedList<TKey,TValue> | SortedList | ImmutableSortedDictionary<TKey,TValue> ImmutableSortedSet<T> |
| A set for mathematical functions | HashSet<T> SortedSet<T> | No recommendation | ImmutableHashSet<T> ImmutableSortedSet<T> |

Algorithmic complexity of collections

When choosing a [collection class](#), it's worth considering potential tradeoffs in performance. Use the following table to reference how various mutable collection types compare in algorithmic complexity to their corresponding immutable counterparts. Often immutable collection types are less performant but provide immutability - which is often a valid comparative benefit.

[Expand table](#)

| Mutable | Amortized | Worst Case | Immutable | Complexity |
|--|-----------|-------------------------|---|------------|
| <code>Stack<T>.Push</code> | O(1) | O(n) | <code>ImmutableStack<T>.Push</code> | O(1) |
| <code>Queue<T>.Enqueue</code> | O(1) | O(n) | <code>ImmutableQueue<T>.Enqueue</code> | O(1) |
| <code>List<T>.Add</code> | O(1) | O(n) | <code>ImmutableList<T>.Add</code> | O(log n) |
| <code>List<T>.Item[Int32]</code> | O(1) | O(1) | <code>ImmutableList<T>.Item[Int32]</code> | O(log n) |
| <code>List<T>.Enumerator</code> | O(n) | O(n) | <code>ImmutableList<T>.Enumerator</code> | O(n) |
| <code>HashSet<T>.Add, lookup</code> | O(1) | O(n) | <code>ImmutableHashSet<T>.Add</code> | O(log n) |
| <code>SortedSet<T>.Add</code> | O(log n) | O(n) | <code>ImmutableSortedSet<T>.Add</code> | O(log n) |
| <code>Dictionary<T>.Add</code> | O(1) | O(n) | <code>ImmutableDictionary<T>.Add</code> | O(log n) |
| <code>Dictionary<T>.lookup</code> | O(1) | O(1) – or strictly O(n) | <code>ImmutableDictionary<T>.lookup</code> | O(log n) |
| <code>SortedDictionary<T>.Add</code> | O(log n) | O(n log n) | <code>ImmutableSortedDictionary<T>.Add</code> | O(log n) |

A `List<T>` can be efficiently enumerated using either a `for` loop or a `foreach` loop. An `ImmutableList<T>`, however, does a poor job inside a `for` loop, due to the $O(\log n)$ time for its indexer. Enumerating an `ImmutableList<T>` using a `foreach` loop is efficient because `ImmutableList<T>` uses a binary tree to store its data instead of an array like `List<T>` uses. An array can be quickly indexed into, whereas a binary tree must be walked down until the node with the desired index is found.

Additionally, `SortedSet<T>` has the same complexity as `ImmutableSortedSet<T>` because they both use binary trees. The significant difference is that `ImmutableSortedSet<T>` uses an immutable binary tree. Since `ImmutableSortedSet<T>` also offers a [System.Collections.Immutable.ImmutableSortedSet<T>.Builder](#) class that allows mutation, you can have both immutability and performance.

Related articles

[+] [Expand table](#)

| Title | Description |
|---|--|
| Selecting a Collection Class | Describes the different collections and helps you select one for your scenario. |
| Commonly Used Collection Types | Describes commonly used generic and non-generic collection types such as System.Array , System.Collections.Generic.List<T> , and System.Collections.Generic.Dictionary< TKey, TValue > . |
| When to Use Generic Collections | Discusses the use of generic collection types. |

| Title | Description |
|---|--|
| Comparisons and Sorts Within Collections | Discusses the use of equality comparisons and sorting comparisons in collections. |
| Sorted Collection Types | Describes sorted collections performance and characteristics. |
| Hashtable and Dictionary Collection Types | Describes the features of generic and non-generic hash-based dictionary types. |
| Thread-Safe Collections | Describes collection types such as <code>System.Collections.Concurrent.BlockingCollection<T></code> and <code>System.Collections.Concurrent.ConcurrentBag<T></code> that support safe and efficient concurrent access from multiple threads. |
| System.Collections.Immutable | Introduces the immutable collections and provides links to the collection types. |

Reference

- [System.Array](#)
- [System.Collections](#)
- [System.Collections.Concurrent](#)
- [System.Collections.Generic](#)
- [System.Collections.Specialized](#)
- [System.Linq](#)
- [System.Collections.Immutable](#)

 [Collaborate with us on GitHub](#)

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

.NET is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

Selecting a Collection Class

Article • 09/15/2021

Be sure to choose your collection class carefully. Using the wrong type can restrict your use of the collection.

ⓘ Important

Avoid using the types in the [System.Collections](#) namespace. The generic and concurrent versions of the collections are recommended because of their greater type safety and other improvements.

Consider the following questions:

- Do you need a sequential list where the element is typically discarded after its value is retrieved?
 - If yes, consider using the [Queue](#) class or the [Queue<T>](#) generic class if you need first-in, first-out (FIFO) behavior. Consider using the [Stack](#) class or the [Stack<T>](#) generic class if you need last-in, first-out (LIFO) behavior. For safe access from multiple threads, use the concurrent versions, [ConcurrentQueue<T>](#) and [ConcurrentStack<T>](#). For immutability, consider the immutable versions, [ImmutableQueue<T>](#) and [ImmutableStack<T>](#).
 - If not, consider using the other collections.
- Do you need to access the elements in a certain order, such as FIFO, LIFO, or random?
 - The [Queue](#) class, as well as the [Queue<T>](#), [ConcurrentQueue<T>](#), and [ImmutableQueue<T>](#) generic classes all offer FIFO access. For more information, see [When to Use a Thread-Safe Collection](#).
 - The [Stack](#) class, as well as the [Stack<T>](#), [ConcurrentStack<T>](#), and [ImmutableStack<T>](#) generic classes all offer LIFO access. For more information, see [When to Use a Thread-Safe Collection](#).
 - The [LinkedList<T>](#) generic class allows sequential access either from the head to the tail, or from the tail to the head.
- Do you need to access each element by index?

- The [ArrayList](#) and [StringCollection](#) classes and the [List<T>](#) generic class offer access to their elements by the zero-based index of the element. For immutability, consider the immutable generic versions, [ImmutableArray<T>](#) and [ImmutableList<T>](#).
 - The [Hashtable](#), [SortedList](#), [ListDictionary](#), and [StringDictionary](#) classes, and the [Dictionary< TKey, TValue >](#) and [SortedDictionary< TKey, TValue >](#) generic classes offer access to their elements by the key of the element. Additionally, there are immutable versions of several corresponding types: [ImmutableHashSet<T>](#), [ImmutableDictionary< TKey, TValue >](#), [ImmutableSortedSet<T>](#), and [ImmutableSortedDictionary< TKey, TValue >](#).
 - The [NameObjectCollectionBase](#) and [NameValuePairCollection](#) classes, and the [KeyedCollection< TKey, TItem >](#) and [SortedList< TKey, TValue >](#) generic classes offer access to their elements by either the zero-based index or the key of the element.
- Will each element contain one value, a combination of one key and one value, or a combination of one key and multiple values?
 - One value: Use any of the collections based on the [IList](#) interface or the [IList<T>](#) generic interface. For an immutable option, consider the [ImmutableList<T>](#) generic interface.
 - One key and one value: Use any of the collections based on the [IDictionary](#) interface or the [IDictionary< TKey, TValue >](#) generic interface. For an immutable option, consider the [ImmutableSet<T>](#) or [ImmutableDictionary< TKey, TValue >](#) generic interfaces.
 - One value with embedded key: Use the [KeyedCollection< TKey, TItem >](#) generic class.
 - One key and multiple values: Use the [NameValuePairCollection](#) class.
 - Do you need to sort the elements differently from how they were entered?
 - The [Hashtable](#) class sorts its elements by their hash codes.
 - The [SortedList](#) class, and the [SortedList< TKey, TValue >](#) and [SortedDictionary< TKey, TValue >](#) generic classes sort their elements by the key. The sort order is based on the implementation of the [IComparer](#) interface for the [SortedList](#) class and on the implementation of the [IComparer<T>](#) generic interface for the [SortedList< TKey, TValue >](#) and [SortedDictionary< TKey, TValue >](#) generic classes. Of the two generic types, [SortedDictionary< TKey, TValue >](#) offers

better performance than `SortedList< TKey, TValue >`, while `SortedList< TKey, TValue >` consumes less memory.

- `ArrayList` provides a `Sort` method that takes an `IComparer` implementation as a parameter. Its generic counterpart, the `List< T >` generic class, provides a `Sort` method that takes an implementation of the `IComparer< T >` generic interface as a parameter.
- Do you need fast searches and retrieval of information?
 - `ListDictionary` is faster than `Hashtable` for small collections (10 items or fewer). The `Dictionary< TKey, TValue >` generic class provides faster lookup than the `SortedDictionary< TKey, TValue >` generic class. The multi-threaded implementation is `ConcurrentDictionary< TKey, TValue >`. `ConcurrentBag< T >` provides fast multi-threaded insertion for unordered data. For more information about both multi-threaded types, see [When to Use a Thread-Safe Collection](#).
- Do you need collections that accept only strings?
 - `StringCollection` (based on `IList`) and `StringDictionary` (based on `IDictionary`) are in the `System.Collections.Specialized` namespace.
 - In addition, you can use any of the generic collection classes in the `System.Collections.Generic` namespace as strongly typed string collections by specifying the `String` class for their generic type arguments. For example, you can declare a variable to be of type `List< String >` or `Dictionary< String, String >`.

LINQ to Objects and PLINQ

LINQ to Objects enables developers to use LINQ queries to access in-memory objects as long as the object type implements `IEnumerable` or `IEnumerable< T >`. LINQ queries provide a common pattern for accessing data, are typically more concise and readable than standard `foreach` loops, and provide filtering, ordering, and grouping capabilities. For more information, see [LINQ to Objects \(C#\)](#) and [LINQ to Objects \(Visual Basic\)](#).

PLINQ provides a parallel implementation of LINQ to Objects that can offer faster query execution in many scenarios, through more efficient use of multi-core computers. For more information, see [Parallel LINQ \(PLINQ\)](#).

See also

- [System.Collections](#)
- [System.Collections.Specialized](#)

- System.Collections.Generic
- Thread-Safe Collections

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

Commonly used collection types

Article • 09/15/2021

Collection types represent different ways to collect data, such as hash tables, queues, stacks, bags, dictionaries, and lists.

All collections are based on the [ICollection](#) or [ICollection<T>](#) interfaces, either directly or indirectly. [IList](#) and [IDictionary](#) and their generic counterparts all derive from these two interfaces.

In collections based on [IList](#) or directly on [ICollection](#), every element contains only a value. These types include:

- [Array](#)
- [ArrayList](#)
- [List<T>](#)
- [Queue](#)
- [ConcurrentQueue<T>](#)
- [Stack](#)
- [ConcurrentStack<T>](#)
- [LinkedList<T>](#)

In collections based on the [IDictionary](#) interface, every element contains both a key and a value. These types include:

- [Hashtable](#)
- [SortedList](#)
- [SortedList<TKey,TValue>](#)
- [Dictionary<TKey,TValue>](#)
- [ConcurrentDictionary<TKey,TValue>](#)

The [KeyedCollection< TKey, TItem >](#) class is unique because it is a list of values with keys embedded within the values. As a result, it behaves both like a list and like a dictionary.

When you need efficient multi-threaded collection access, use the generic collections in the [System.Collections.Concurrent](#) namespace.

The [Queue](#) and [Queue<T>](#) classes provide first-in-first-out lists. The [Stack](#) and [Stack<T>](#) classes provide last-in-first-out lists.

Strong typing

Generic collections are the best solution to strong typing. For example, adding an element of any type other than an `Int32` to a `List<Int32>` collection causes a compile-time error. However, if your language does not support generics, the `System.Collections` namespace includes abstract base classes that you can extend to create collection classes that are strongly typed. These base classes include:

- `CollectionBase`
- `ReadOnlyCollectionBase`
- `DictionaryBase`

How collections vary

Collections vary in how they store, sort, and compare elements, and how they perform searches.

The `SortedList` class and the `SortedList<TKey,TValue>` generic class provide sorted versions of the `Hashtable` class and the `Dictionary<TKey,TValue>` generic class.

All collections use zero-based indexes except `Array`, which allows arrays that are not zero-based.

You can access the elements of a `SortedList` or a `KeyedCollection<TKey,TItem>` by either the key or the element's index. You can only access the elements of a `Hashtable` or a `Dictionary<TKey,TValue>` by the element's key.

Use LINQ with collection types

The LINQ to Objects feature provides a common pattern for accessing in-memory objects of any type that implements `IEnumerable` or `IEnumerable<T>`. LINQ queries have several benefits over standard constructs like `foreach` loops:

- They are concise and easier to understand.
- They can filter, order, and group data.
- They can improve performance.

For more information, see [LINQ to Objects \(C#\)](#), [LINQ to Objects \(Visual Basic\)](#), and [Parallel LINQ \(PLINQ\)](#).

Related topics

| Title | Description |
|---|---|
| Collections and Data Structures | Discusses the various collection types available in .NET, including stacks, queues, lists, arrays, and dictionaries. |
| Hashtable and Dictionary Collection Types | Describes the features of generic and nongeneric hash-based dictionary types. |
| Sorted Collection Types | Describes classes that provide sorting functionality for lists and sets. |
| Generics | Describes the generics feature, including the generic collections, delegates, and interfaces provided by .NET. Provides links to feature documentation for C#, Visual Basic, and Visual C++, and to supporting technologies such as reflection. |

Reference

[System.Collections](#)

[System.Collections.Generic](#)

[System.Collections.ICollection](#)

[System.Collections.Generic.ICollection<T>](#)

[System.Collections_IList](#)

[System.Collections.Generic_IList<T>](#)

[System.Collections.IDictionary](#)

[System.Collections.Generic.IDictionary< TKey, TValue >](#)

 Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback [here](#).

 [Open a documentation issue](#)

 [Provide product feedback](#)

When to use generic collections

Article • 09/15/2021

Using generic collections gives you the automatic benefit of type safety without having to derive from a base collection type and implement type-specific members. Generic collection types also generally perform better than the corresponding nongeneric collection types (and better than types that are derived from nongeneric base collection types) when the collection elements are value types, because with generics, there's no need to box the elements.

For programs that target .NET Standard 1.0 or later, use the generic collection classes in the [System.Collections.Concurrent](#) namespace when multiple threads might be adding or removing items from the collection concurrently. Additionally, when immutability is desired, consider the generic collection classes in the [System.Collections.Immutable](#) namespace.

The following generic types correspond to existing collection types:

- [List<T>](#) is the generic class that corresponds to [ArrayList](#).
- [Dictionary< TKey, TValue >](#) and [ConcurrentDictionary< TKey, TValue >](#) are the generic classes that correspond to [Hashtable](#).
- [Collection<T>](#) is the generic class that corresponds to [CollectionBase](#). [Collection<T>](#) can be used as a base class, but unlike [CollectionBase](#), it is not abstract, which makes it much easier to use.
- [ReadOnlyCollection<T>](#) is the generic class that corresponds to [ReadOnlyCollectionBase](#). [ReadOnlyCollection<T>](#) is not abstract and has a constructor that makes it easy to expose an existing [List<T>](#) as a read-only collection.
- The [Queue<T>](#), [ConcurrentQueue<T>](#), [ImmutableQueue<T>](#), [ImmutableArray<T>](#), [SortedList< TKey, TValue >](#), and [ImmutableSortedSet< T >](#) generic classes correspond to the respective nongeneric classes with the same names.

Additional Types

Several generic collection types do not have nongeneric counterparts. They include the following:

- `LinkedList<T>` is a general-purpose linked list that provides O(1) insertion and removal operations.
- `SortedDictionary< TKey, TValue >` is a sorted dictionary with O(log n) insertion and retrieval operations, which makes it a useful alternative to `SortedList< TKey, TValue >`.
- `KeyedCollection< TKey, TItem >` is a hybrid between a list and a dictionary, which provides a way to store objects that contain their own keys.
- `BlockingCollection<T>` implements a collection class with bounding and blocking functionality.
- `ConcurrentBag<T>` provides fast insertion and removal of unordered elements.

Immutable builders

When you desire immutability functionality in your app, the `System.Collections.Immutable` namespace offers generic collection types you can use. All of the immutable collection types offer `Builder` classes that can optimize performance when you're performing multiple mutations. The `Builder` class batches operations in a mutable state. When all mutations have been completed, call the `ToImmutable` method to "freeze" all nodes and create an immutable generic collection, for example, an `ImmutableList<T>`.

The `Builder` object can be created by calling the nongeneric `CreateBuilder()` method. From a `Builder` instance, you can call `ToImmutable()`. Likewise, from the `Immutable*` collection, you can call `ToBuilder()` to create a builder instance from the generic immutable collection. The following are the various `Builder` types.

- `ImmutableArray<T>.Builder`
- `ImmutableDictionary< TKey, TValue >.Builder`
- `ImmutableHashSet<T>.Builder`
- `ImmutableList<T>.Builder`
- `ImmutableSortedDictionary< TKey, TValue >.Builder`
- `ImmutableSortedSet<T>.Builder`

LINQ to Objects

The LINQ to Objects feature enables you to use LINQ queries to access in-memory objects as long as the object type implements the `System.Collections.IEnumerable` or `System.Collections.Generic.IEnumerable<T>` interface. LINQ queries provide a common

pattern for accessing data; are typically more concise and readable than standard `foreach` loops; and provide filtering, ordering, and grouping capabilities. LINQ queries can also improve performance. For more information, see [LINQ to Objects \(C#\)](#), [LINQ to Objects \(Visual Basic\)](#), and [Parallel LINQ \(PLINQ\)](#).

Additional Functionality

Some of the generic types have functionality that is not found in the nongeneric collection types. For example, the `List<T>` class, which corresponds to the nongeneric `ArrayList` class, has a number of methods that accept generic delegates, such as the `Predicate<T>` delegate that allows you to specify methods for searching the list, the `Action<T>` delegate that represents methods that act on each element of the list, and the `Converter<TInput,TOutput>` delegate that lets you define conversions between types.

The `List<T>` class allows you to specify your own `IComparer<T>` generic interface implementations for sorting and searching the list. The `SortedDictionary< TKey, TValue >` and `SortedList< TKey, TValue >` classes also have this capability. In addition, these classes let you specify comparers when the collection is created. In similar fashion, the `Dictionary< TKey, TValue >` and `KeyedCollection< TKey, TItem >` classes let you specify your own equality comparers.

See also

- [Collections and Data Structures](#)
- [Commonly Used Collection Types](#)
- [Generics](#)

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback [here](#).

 [Open a documentation issue](#)

 [Provide product feedback](#)

Comparisons and sorts within collections

Article • 09/15/2021

The [System.Collections](#) classes perform comparisons in almost all the processes involved in managing collections, whether searching for the element to remove or returning the value of a key-and-value pair.

Collections typically utilize an equality comparer and/or an ordering comparer. Two constructs are used for comparisons.

Check for equality

Methods such as `Contains`, `IndexOf`, `LastIndexOf`, and `Remove` use an equality comparer for the collection elements. If the collection is generic, then items are compared for equality according to the following guidelines:

- If type T implements the [IEquatable<T>](#) generic interface, then the equality comparer is the `Equals` method of that interface.
- If type T does not implement [IEquatable<T>](#), `Object.Equals` is used.

In addition, some constructor overloads for dictionary collections accept an [IEqualityComparer<T>](#) implementation, which is used to compare keys for equality. For an example, see the [Dictionary<TKey,TValue>](#) constructor.

Determine sort order

Methods such as `BinarySearch` and `Sort` use an ordering comparer for the collection elements. The comparisons can be between elements of the collection, or between an element and a specified value. For comparing objects, there is the concept of a `default comparer` and an `explicit comparer`.

The default comparer relies on at least one of the objects being compared to implement the [IComparable](#) interface. It is a good practice to implement [IComparable](#) on all classes which are used as values in a list collection or as keys in a dictionary collection. For a generic collection, equality comparison is determined according to the following:

- If type T implements the [System.IComparable<T>](#) generic interface, then the default comparer is the `IComparable<T>.CompareTo(T)` method of that interface

- If type T implements the non-generic [System.IComparable](#) interface, then the default comparer is the [IComparable.CompareTo\(Object\)](#) method of that interface.
- If type T doesn't implement either interface, then there is no default comparer, and a comparer or comparison delegate must be provided explicitly.

To provide explicit comparisons, some methods accept an [IComparer](#) implementation as a parameter. For example, the [List<T>.Sort](#) method accepts an [System.Collections.Generic.IComparer<T>](#) implementation.

The current culture setting of the system can affect the comparisons and sorts within a collection. By default, the comparisons and sorts in the [Collections](#) classes are culture-sensitive. To ignore the culture setting and therefore obtain consistent comparison and sorting results, use the [InvariantCulture](#) with member overloads that accept a [CultureInfo](#). For more information, see [Perform culture-insensitive string operations in collections](#) and [Perform culture-insensitive string operations in arrays](#).

Equality and sort example

The following code demonstrates an implementation of [IEquatable<T>](#) and [IComparable<T>](#) on a simple business object. In addition, when the object is stored in a list and sorted, you will see that calling the [Sort\(\)](#) method results in the use of the default comparer for the [Part](#) type, and the [Sort\(Comparison<T>\)](#) method implemented by using an anonymous method.

C#

```
using System;
using System.Collections.Generic;

// Simple business object. A PartId is used to identify the
// type of part but the part name can change.
public class Part : IEquatable<Part>, IComparable<Part>
{
    public string PartName { get; set; }

    public int PartId { get; set; }

    public override string ToString() =>
        $"ID: {PartId}    Name: {PartName}";

    public override bool Equals(object obj) =>
        (obj is Part part)
            ? Equals(part)
            : false;

    public int SortByNameAscending(string name1, string name2) =>
        string.Compare(name1, name2);
}
```

```
name1?.CompareTo(name2) ?? 1;

// Default comparer for Part type.
// A null value means that this object is greater.
public int CompareTo(Part comparePart) =>
    comparePart == null ? 1 : PartId.CompareTo(comparePart.PartId);

public override int GetHashCode() => PartId;

public bool Equals(Part other) =>
    other is null ? false : PartId.Equals(other.PartId);

// Should also override == and != operators.
}

public class Example
{
    public static void Main()
    {
        // Create a list of parts.
        var parts = new List<Part>
        {
            // Add parts to the list.
            new Part { PartName = "regular seat", PartId = 1434 },
            new Part { PartName = "crank arm", PartId = 1234 },
            new Part { PartName = "shift lever", PartId = 1634 },
            // Name intentionally left null.
            new Part { PartId = 1334 },
            new Part { PartName = "banana seat", PartId = 1444 },
            new Part { PartName = "cassette", PartId = 1534 }
        };

        // Write out the parts in the list. This will call the overridden
        // ToString method in the Part class.
        Console.WriteLine("\nBefore sort:");
        parts.ForEach(Console.WriteLine);

        // Call Sort on the list. This will use the
        // default comparer, which is the Compare method
        // implemented on Part.
        parts.Sort();

        Console.WriteLine("\nAfter sort by part number:");
        parts.ForEach(Console.WriteLine);

        // This shows calling the Sort(Comparison<T> comparison) overload
using
        // a lambda expression as the Comparison<T> delegate.
        // This method treats null as the lesser of two values.
        parts.Sort((Part x, Part y) =>
            x.PartName == null && y.PartName == null
                ? 0
                : x.PartName == null
                    ? -1
                    : y.PartName == null
```

```

        ? 1
        : x.PartName.CompareTo(y.PartName));

Console.WriteLine("\nAfter sort by name:");
parts.ForEach(Console.WriteLine);

/*
    Before sort:
ID: 1434  Name: regular seat
ID: 1234  Name: crank arm
ID: 1634  Name: shift lever
ID: 1334  Name:
ID: 1444  Name: banana seat
ID: 1534  Name: cassette

    After sort by part number:
ID: 1234  Name: crank arm
ID: 1334  Name:
ID: 1434  Name: regular seat
ID: 1444  Name: banana seat
ID: 1534  Name: cassette
ID: 1634  Name: shift lever

    After sort by name:
ID: 1334  Name:
ID: 1444  Name: banana seat
ID: 1534  Name: cassette
ID: 1234  Name: crank arm
ID: 1434  Name: regular seat
ID: 1634  Name: shift lever

*/
}

}

```

See also

- [IComparer](#)
- [IEquatable<T>](#)
- [IComparer<T>](#)
- [IComparable](#)
- [IComparable<T>](#)



Collaborate with us on
GitHub



.NET feedback

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET is an open source project.
Select a link to provide feedback:

-  [Open a documentation issue](#)
-  [Provide product feedback](#)

Sorted Collection Types

Article • 09/15/2021

The [System.Collections.SortedList](#) class, the [System.Collections.Generic.SortedList<TKey,TValue>](#) generic class, and the [System.Collections.Generic.SortedDictionary<TKey,TValue>](#) generic class are similar to the [Hashtable](#) class and the [Dictionary<TKey,TValue>](#) generic class in that they implement the [IDictionary](#) interface, but they maintain their elements in sort order by key, and they do not have the O(1) insertion and retrieval characteristic of hash tables. The three classes have several features in common:

- All three classes implement the [System.Collections.IDictionary](#) interface. The two generic classes also implement the [System.Collections.Generic.IDictionary<TKey,TValue>](#) generic interface.
- Each element is a key/value pair for enumeration purposes.

ⓘ Note

The nongeneric [SortedList](#) class returns [DictionaryEntry](#) objects when enumerated, although the two generic types return [KeyValuePair<TKey,TValue>](#) objects.

- Elements are sorted according to a [System.Collections.IComparer](#) implementation (for nongeneric [SortedList](#)) or a [System.Collections.Generic.IComparer<T>](#) implementation (for the two generic classes).
- Each class provides properties that return collections containing only the keys or only the values.

The following table lists some of the differences between the two sorted list classes and the [SortedDictionary<TKey,TValue>](#) class.

| SortedList nongeneric class and SortedList<TKey,TValue> generic class | SortedDictionary<TKey,TValue> generic class |
|---|--|
| The properties that return keys and values are indexed, allowing efficient indexed retrieval. | No indexed retrieval. |
| Retrieval is O(log n). | Retrieval is O(log n). |
| Insertion and removal are generally O(n); however, insertion is O(log n) for data that are already in sort | Insertion and removal are O(log n). |

| SortedList nongeneric class and SortedList<TKey,TValue> generic class | SortedDictionary<TKey,TValue> generic class |
|--|---|
| order, so that each element is added to the end of the list. (This assumes that a resize is not required.) | |
| Uses less memory than a SortedDictionary<TKey,TValue> . | Uses more memory than the SortedList nongeneric class and the SortedList<TKey,TValue> generic class . |

For sorted lists or dictionaries that must be accessible concurrently from multiple threads, you can add sorting logic to a class that derives from [ConcurrentDictionary<TKey,TValue>](#). When considering immutability, the following corresponding immutable types follow similar sorting semantics: [ImmutableSortedSet<T>](#) and [ImmutableSortedDictionary<TKey,TValue>](#).

ⓘ Note

For values that contain their own keys (for example, employee records that contain an employee ID number), you can create a keyed collection that has some characteristics of a list and some characteristics of a dictionary by deriving from the [KeyedCollection<TKey,TItem>](#) generic class.

Starting with .NET Framework 4, the [SortedSet<T>](#) class provides a self-balancing tree that maintains data in sorted order after insertions, deletions, and searches. This class and the [HashSet<T>](#) class implement the [ISet<T>](#) interface.

See also

- [System.Collections.IDictionary](#)
- [System.Collections.Generic.IDictionary<TKey,TValue>](#)
- [ConcurrentDictionary<TKey,TValue>](#)
- [Commonly Used Collection Types](#)

⌚ Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

⌚ Open a documentation issue

more information, see [our contributor guide](#).

 [Provide product feedback](#)

Hashtable and Dictionary Collection Types

Article • 09/15/2021

The [System.Collections.Hashtable](#) class, and the [System.Collections.Generic.Dictionary<TKey,TValue>](#) and [System.Collections.Concurrent.ConcurrentDictionary<TKey,TValue>](#) generic classes, implement the [System.Collections.IDictionary](#) interface. The [Dictionary<TKey,TValue>](#) generic class also implements the [IDictionary<TKey,TValue>](#) generic interface. Therefore, each element in these collections is a key-and-value pair.

A [Hashtable](#) object consists of buckets that contain the elements of the collection. A bucket is a virtual subgroup of elements within the [Hashtable](#), which makes searching and retrieving easier and faster than in most collections. Each bucket is associated with a hash code, which is generated using a hash function and is based on the key of the element.

The generic [HashSet<T>](#) class is an unordered collection for containing unique elements.

A hash function is an algorithm that returns a numeric hash code based on a key. The key is the value of some property of the object being stored. A hash function must always return the same hash code for the same key. It is possible for a hash function to generate the same hash code for two different keys, but a hash function that generates a unique hash code for each unique key results in better performance when retrieving elements from the hash table.

Each object that is used as an element in a [Hashtable](#) must be able to generate a hash code for itself by using an implementation of the [GetHashCode](#) method. However, you can also specify a hash function for all elements in a [Hashtable](#) by using a [Hashtable](#) constructor that accepts an [IHashCodeProvider](#) implementation as one of its parameters.

When an object is added to a [Hashtable](#), it is stored in the bucket that is associated with the hash code that matches the object's hash code. When a value is being searched for in the [Hashtable](#), the hash code is generated for that value, and the bucket associated with that hash code is searched.

For example, a hash function for a string might take the ASCII codes of each character in the string and add them together to generate a hash code. The string "picnic" would have a hash code that is different from the hash code for the string "basket"; therefore,

the strings "picnic" and "basket" would be in different buckets. In contrast, "stressed" and "desserts" would have the same hash code and would be in the same bucket.

The [Dictionary<TKey,TValue>](#) and [ConcurrentDictionary<TKey,TValue>](#) classes have the same functionality as the [Hashtable](#) class. A [Dictionary<TKey,TValue>](#) of a specific type (other than [Object](#)) provides better performance than a [Hashtable](#) for value types. This is because the elements of [Hashtable](#) are of type [Object](#); therefore, boxing and unboxing typically occur when you store or retrieve a value type. The [ConcurrentDictionary<TKey,TValue>](#) class should be used when multiple threads might be accessing the collection simultaneously.

See also

- [Hashtable](#)
- [IDictionary](#)
- [IHashCodeProvider](#)
- [Dictionary<TKey,TValue>](#)
- [System.Collections.Generic.IDictionary<TKey,TValue>](#)
- [System.Collections.Concurrent.ConcurrentDictionary<TKey,TValue>](#)
- [Commonly Used Collection Types](#)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



.NET feedback

The .NET documentation is open source. Provide feedback [here](#).

 [Open a documentation issue](#)

 [Provide product feedback](#)

Thread-safe collections

Article • 01/27/2023

The [System.Collections.Concurrent](#) namespace includes several collection classes that are both thread-safe and scalable. Multiple threads can safely and efficiently add or remove items from these collections, without requiring additional synchronization in user code. When you write new code, use the concurrent collection classes to write multiple threads to the collection concurrently. If you're only reading from a shared collection, then you can use the classes in the [System.Collections.Generic](#) namespace.

System.Collections and System.Collections.Generic

The collection classes in the [System.Collections](#) namespace include [ArrayList](#) and [Hashtable](#). These classes provide some thread safety through the `Synchronized` property, which returns a thread-safe wrapper around the collection. The wrapper works by locking the entire collection on every add or remove operation. Therefore, each thread that's attempting to access the collection must wait for its turn to take the one lock. This process isn't scalable and can cause significant performance degradation for large collections. Also, the design isn't protected from race conditions. For more information, see [Synchronization in Generic Collections](#).

The collection classes in the [System.Collections.Generic](#) namespace include [List<T>](#) and [Dictionary< TKey, TValue >](#). These classes provide improved type safety and performance compared to the [System.Collections](#) classes. However, the [System.Collections.Generic](#) classes don't provide any thread synchronization; user code must provide all synchronization when items are added or removed on multiple threads concurrently.

We recommend using the concurrent collections classes in the [System.Collections.Concurrent](#) namespace because they provide type safety and also more efficient and complete thread safety.

Fine-grained locking and lock-free mechanisms

Some of the concurrent collection types use lightweight synchronization mechanisms such as [SpinLock](#), [SpinWait](#), [SemaphoreSlim](#), and [CountdownEvent](#). These synchronization types typically use *busy spinning* for brief periods before they put the thread into a true `Wait` state. When wait times are expected to be short, spinning is far less computationally expensive than waiting, which involves an expensive kernel

transition. For collection classes that use spinning, this efficiency means that multiple threads can add and remove items at a high rate. For more information about spinning versus blocking, see [SpinLock](#) and [SpinWait](#).

The [ConcurrentQueue<T>](#) and [ConcurrentStack<T>](#) classes don't use locks at all. Instead, they rely on [Interlocked](#) operations to achieve thread safety.

① Note

Because the concurrent collections classes support [ICollection](#), they provide implementations for the [IsSynchronized](#) and [SyncRoot](#) properties, even though these properties are irrelevant. `IsSynchronized` always returns `false` and, `SyncRoot` is always `null` (`Nothing` in Visual Basic).

The following table lists the collection types in the [System.Collections.Concurrent](#) namespace:

| Type | Description |
|---|--|
| BlockingCollection<T> | Provides bounding and blocking functionality for any type that implements IProducerConsumerCollection<T> . For more information, see BlockingCollection Overview . |
| ConcurrentDictionary<TKey,TValue> | Thread-safe implementation of a dictionary of key-value pairs. |
| ConcurrentQueue<T> | Thread-safe implementation of a FIFO (first-in, first-out) queue. |
| ConcurrentStack<T> | Thread-safe implementation of a LIFO (last-in, first-out) stack. |
| ConcurrentBag<T> | Thread-safe implementation of an unordered collection of elements. |
| IProducerConsumerCollection<T> | The interface that a type must implement to be used in a BlockingCollection . |

Related articles

| Title | Description |
|---|---|
| BlockingCollection Overview | Describes the functionality provided by the BlockingCollection<T> type. |

| Title | Description |
|---|--|
| How to: Add and Remove Items from a ConcurrentDictionary | Describes how to add and remove elements from a <code>ConcurrentDictionary<TKey, TValue></code> |
| How to: Add and Take Items Individually from a BlockingCollection | Describes how to add and retrieve items from a blocking collection without using the read-only enumerator. |
| How to: Add Bounding and Blocking Functionality to a Collection | Describes how to use any collection class as the underlying storage mechanism for an <code>IProducerConsumerCollection<T></code> collection. |
| How to: Use ForEach to Remove Items in a BlockingCollection | Describes how to use <code>foreach</code> (<code>For Each</code> in Visual Basic) to remove all items in a blocking collection. |
| How to: Use Arrays of Blocking Collections in a Pipeline | Describes how to use multiple blocking collections at the same time to implement a pipeline. |
| How to: Create an Object Pool by Using a ConcurrentBag | Shows how to use a concurrent bag to improve performance in scenarios where you can reuse objects instead of continually creating new ones. |

Reference

- [System.Collections.Concurrent](#)

 Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback [here](#).

 [Open a documentation issue](#)

 [Provide product feedback](#)

Delegates and lambdas

Article • 01/05/2022

A delegate defines a type that represents references to methods that have a particular parameter list and return type. A method (static or instance) whose parameter list and return type match can be assigned to a variable of that type, then called directly (with the appropriate arguments) or passed as an argument itself to another method and then called. The following example demonstrates delegate use.

C#

```
using System;
using System.Linq;

public class Program
{
    public delegate string Reverse(string s);

    static string ReverseString(string s)
    {
        return new string(s.Reverse().ToArray());
    }

    static void Main(string[] args)
    {
        Reverse rev = ReverseString;

        Console.WriteLine(rev("a string"));
    }
}
```

- The `public delegate string Reverse(string s);` line creates a delegate type of a method that takes a string parameter and then returns a string parameter.
- The `static string ReverseString(string s)` method, which has the exact same parameter list and return type as the defined delegate type, implements the delegate.
- The `Reverse rev = ReverseString;` line shows that you can assign a method to a variable of the corresponding delegate type.
- The `Console.WriteLine(rev("a string"));` line demonstrates how to use a variable of a delegate type to invoke the delegate.

In order to streamline the development process, .NET includes a set of delegate types that programmers can reuse and not have to create new types. These types are `Func<>`, `Action<>` and `Predicate<>`, and they can be used without having to define new delegate

types. There are some differences between the three types that have to do with the way they were intended to be used:

- `Action<>` is used when there is a need to perform an action using the arguments of the delegate. The method it encapsulates does not return a value.
- `Func<>` is used usually when you have a transformation on hand, that is, you need to transform the arguments of the delegate into a different result. Projections are a good example. The method it encapsulates returns a specified value.
- `Predicate<>` is used when you need to determine if the argument satisfies the condition of the delegate. It can also be written as a `Func<T, bool>`, which means the method returns a boolean value.

We can now take our example above and rewrite it using the `Func<>` delegate instead of a custom type. The program will continue running exactly the same.

C#

```
using System;
using System.Linq;

public class Program
{
    static string ReverseString(string s)
    {
        return new string(s.Reverse().ToArray());
    }

    static void Main(string[] args)
    {
        Func<string, string> rev = ReverseString;

        Console.WriteLine(rev("a string"));
    }
}
```

For this simple example, having a method defined outside of the `Main` method seems a bit superfluous. .NET Framework 2.0 introduced the concept of *anonymous delegates*, which let you create "inline" delegates without having to specify any additional type or method.

In the following example, an anonymous delegate filters a list to just the even numbers and then prints them to the console.

C#

```
using System;
using System.Collections.Generic;
```

```

public class Program
{
    public static void Main(string[] args)
    {
        List<int> list = new List<int>();

        for (int i = 1; i <= 100; i++)
        {
            list.Add(i);
        }

        List<int> result = list.FindAll(
            delegate (int no)
            {
                return (no % 2 == 0);
            }
        );

        foreach (var item in result)
        {
            Console.WriteLine(item);
        }
    }
}

```

As you can see, the body of the delegate is just a set of expressions, as any other delegate. But instead of it being a separate definition, we've introduced it *ad hoc* in our call to the `List<T>.FindAll` method.

However, even with this approach, there is still much code that we can throw away. This is where *lambda expressions* come into play. Lambda expressions, or just "lambdas" for short, were introduced in C# 3.0 as one of the core building blocks of Language Integrated Query (LINQ). They are just a more convenient syntax for using delegates. They declare a parameter list and method body, but don't have a formal identity of their own, unless they are assigned to a delegate. Unlike delegates, they can be directly assigned as the right-hand side of event registration or in various LINQ clauses and methods.

Since a lambda expression is just another way of specifying a delegate, we should be able to rewrite the above sample to use a lambda expression instead of an anonymous delegate.

C#

```

using System;
using System.Collections.Generic;

public class Program

```

```
{  
    public static void Main(string[] args)  
    {  
        List<int> list = new List<int>();  
  
        for (int i = 1; i <= 100; i++)  
        {  
            list.Add(i);  
        }  
  
        List<int> result = list.FindAll(i => i % 2 == 0);  
  
        foreach (var item in result)  
        {  
            Console.WriteLine(item);  
        }  
    }  
}
```

In the preceding example, the lambda expression used is `i => i % 2 == 0`. Again, it is just a convenient syntax for using delegates. What happens under the covers is similar to what happens with the anonymous delegate.

Again, lambdas are just delegates, which means that they can be used as an event handler without any problems, as the following code snippet illustrates.

C#

```
public MainWindow()  
{  
    InitializeComponent();  
  
    Loaded += (o, e) =>  
    {  
        this.Title = "Loaded";  
    };  
}
```

The `+=` operator in this context is used to subscribe to an [event](#). For more information, see [How to subscribe to and unsubscribe from events](#).

Further reading and resources

- [Delegates](#)
- [Lambda expressions](#)

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

System.Enum class

Article • 01/08/2024

This article provides supplementary remarks to the reference documentation for this API.

An enumeration is a set of named constants whose underlying type is any integral type. If no underlying type is explicitly declared, [Int32](#) is used. [Enum](#) is the base class for all enumerations in .NET. Enumeration types are defined by the `enum` keyword in C#, the `Enum ... End Enum` construct in Visual Basic, and the `type` keyword in F#.

[Enum](#) provides methods for comparing instances of this class, converting the value of an instance to its string representation, converting the string representation of a number to an instance of this class, and creating an instance of a specified enumeration and value.

You can also treat an enumeration as a bit field. For more information, see the [Non-exclusive members and the Flags attribute](#) section and [FlagsAttribute](#).

Create an enumeration type

Programming languages typically provide syntax to declare an enumeration that consists of a set of named constants and their values. The following example illustrates the syntax used by C#, F#, and Visual Basic to define an enumeration. It creates an enumeration named `ArrivalStatus` that has three members: `ArrivalStatus.Early`, `ArrivalStatus.OnTime`, and `ArrivalStatus.Late`. Note that in all cases, the enumeration does not explicitly inherit from [Enum](#); the inheritance relationship is handled implicitly by the compiler.

C#

```
public enum ArrivalStatus { Unknown=-3, Late=-1, OnTime=0, Early=1 };
```

⚠ Warning

You should never create an enumeration type whose underlying type is non-integral or [Char](#). Although you can create such an enumeration type by using reflection, method calls that use the resulting type are unreliable and may also throw additional exceptions.

Instantiate an enumeration type

You can instantiate an enumeration type just as you instantiate any other value type: by declaring a variable and assigning one of the enumeration's constants to it. The following example instantiates an `ArrivalStatus` whose value is `ArrivalStatus.OnTime`.

C#

```
public class Example
{
    public static void Main()
    {
        ArrivalStatus status = ArrivalStatus.OnTime;
        Console.WriteLine("Arrival Status: {0} ({0:D})", status);
    }
}
// The example displays the following output:
//      Arrival Status: OnTime (0)
```

You can also instantiate an enumeration value in the following ways:

- By using a particular programming language's features to cast (as in C#) or convert (as in Visual Basic) an integer value to an enumeration value. The following example creates an `ArrivalStatus` object whose value is `ArrivalStatus.Early` in this way.

C#

```
ArrivalStatus status2 = (ArrivalStatus)1;
Console.WriteLine("Arrival Status: {0} ({0:D})", status2);
// The example displays the following output:
//      Arrival Status: Early (1)
```

- By calling its implicit parameterless constructor. As the following example shows, in this case the underlying value of the enumeration instance is 0. However, this is not necessarily the value of a valid constant in the enumeration.

C#

```
ArrivalStatus status1 = new ArrivalStatus();
Console.WriteLine("Arrival Status: {0} ({0:D})", status1);
// The example displays the following output:
//      Arrival Status: OnTime (0)
```

- By calling the `Parse` or `TryParse` method to parse a string that contains the name of a constant in the enumeration. For more information, see the [Parse enumeration](#)

[values](#) section.

- By calling the [ToObject](#) method to convert an integral value to an enumeration type. For more information, see the [Perform conversions](#) section.

Enumeration best practices

We recommend that you use the following best practices when you define enumeration types:

- If you have not defined an enumeration member whose value is 0, consider creating a `None` enumerated constant. By default, the memory used for the enumeration is initialized to zero by the common language runtime. Consequently, if you do not define a constant whose value is zero, the enumeration will contain an illegal value when it is created.
- If there is an obvious default case that your application has to represent, consider using an enumerated constant whose value is zero to represent it. If there is no default case, consider using an enumerated constant whose value is zero to specify the case that is not represented by any of the other enumerated constants.
- Do not specify enumerated constants that are reserved for future use.
- When you define a method or property that takes an enumerated constant as a value, consider validating the value. The reason is that you can cast a numeric value to the enumeration type even if that numeric value is not defined in the enumeration.

Additional best practices for enumeration types whose constants are bit fields are listed in the [Non-exclusive members and the Flags attribute](#) section.

Perform operations with enumerations

You cannot define new methods when you are creating an enumeration. However, an enumeration type inherits a complete set of static and instance methods from the [Enum](#) class. The following sections survey most of these methods, in addition to several other methods that are commonly used when working with enumeration values.

Perform conversions

You can convert between an enumeration member and its underlying type by using a casting (in C# and F#), or conversion (in Visual Basic) operator. In F#, the `enum` function

is also used. The following example uses casting or conversion operators to perform conversions both from an integer to an enumeration value and from an enumeration value to an integer.

```
C#
```

```
int value3 = 2;
ArrivalStatus status3 = (ArrivalStatus)value3;

int value4 = (int)status3;
```

The [Enum](#) class also includes a [ToObject](#) method that converts a value of any integral type to an enumeration value. The following example uses the [ToObject\(Type, Int32\)](#) method to convert an [Int32](#) to an [ArrivalStatus](#) value. Note that, because the [ToObject](#) returns a value of type [Object](#), the use of a casting or conversion operator may still be necessary to cast the object to the enumeration type.

```
C#
```

```
int number = -1;
ArrivalStatus arrived =
(ArrivalStatus)ArrivalStatus.ToObject(typeof(ArrivalStatus), number);
```

When converting an integer to an enumeration value, it is possible to assign a value that is not actually a member of the enumeration. To prevent this, you can pass the integer to the [IsDefined](#) method before performing the conversion. The following example uses this method to determine whether the elements in an array of integer values can be converted to [ArrivalStatus](#) values.

```
C#
```

```
using System;

public class Example3
{
    public static void Main()
    {
        int[] values = { -3, -1, 0, 1, 5, Int32.MaxValue };
        foreach (var value in values)
        {
            ArrivalStatus status;
            if (Enum.IsDefined(typeof(ArrivalStatus), value))
                status = (ArrivalStatus)value;
            else
                status = ArrivalStatus.Unknown;
            Console.WriteLine("Converted {0:N0} to {1}", value, status);
        }
    }
}
```

```
        }
    }
// The example displays the following output:
//      Converted -3 to Unknown
//      Converted -1 to Late
//      Converted 0 to OnTime
//      Converted 1 to Early
//      Converted 5 to Unknown
//      Converted 2,147,483,647 to Unknown
```

Although the [Enum](#) class provides explicit interface implementations of the [IConvertible](#) interface for converting from an enumeration value to an integral type, you should use the methods of the [Convert](#) class, such as [ToInt32](#), to perform these conversions. The following example illustrates how you can use the [GetUnderlyingType](#) method along with the [Convert.ChangeType](#) method to convert an enumeration value to its underlying type. Note that this example does not require the underlying type of the enumeration to be known at compile time.

C#

```
ArrivalStatus status = ArrivalStatus.Early;
var number = Convert.ChangeType(status,
Enum.GetUnderlyingType(typeof(ArrivalStatus)));
Console.WriteLine("Converted {0} to {1}", status, number);
// The example displays the following output:
//      Converted Early to 1
```

Parse enumeration values

The [Parse](#) and [TryParse](#) methods allow you to convert the string representation of an enumeration value to that value. The string representation can be either the name or the underlying value of an enumeration constant. Note that the parsing methods will successfully convert string representations of numbers that are not members of a particular enumeration if the strings can be converted to a value of the enumeration's underlying type. To prevent this, the [IsDefined](#) method can be called to ensure that the result of the parsing method is a valid enumeration value. The example illustrates this approach and demonstrates calls to both the [Parse\(Type, String\)](#) and [Enum.TryParse<TEnum>\(String, TEnum\)](#) methods. Note that the non-generic parsing method returns an object that you may have to cast (in C# and F#) or convert (in Visual Basic) to the appropriate enumeration type.

C#

```
string number = "-1";
string name = "Early";
```

```

try
{
    ArrivalStatus status1 = (ArrivalStatus)Enum.Parse(typeof(ArrivalStatus),
number);
    if (!(Enum.IsDefined(typeof(ArrivalStatus), status1)))
        status1 = ArrivalStatus.Unknown;
    Console.WriteLine("Converted '{0}' to {1}", number, status1);
}
catch (FormatException)
{
    Console.WriteLine("Unable to convert '{0}' to an ArrivalStatus value.",
number);
}

ArrivalStatus status2;
if (Enum.TryParse<ArrivalStatus>(name, out status2))
{
    if (!(Enum.IsDefined(typeof(ArrivalStatus), status2)))
        status2 = ArrivalStatus.Unknown;
    Console.WriteLine("Converted '{0}' to {1}", name, status2);
}
else
{
    Console.WriteLine("Unable to convert '{0}' to an ArrivalStatus value.",
number);
}
// The example displays the following output:
//      Converted '-1' to Late
//      Converted 'Early' to Early

```

Format enumeration values

You can convert enumeration values to their string representations by calling the static [Format](#) method, as well as the overloads of the instance [ToString](#) method. You can use a format string to control the precise way in which an enumeration value is represented as a string. For more information, see [Enumeration Format Strings](#). The following example uses each of the supported enumeration format strings ("G" or "g", "D" or "d", "X" or "x", and "F" or "f") to convert a member of the `ArrivalStatus` enumeration to its string representations.

C#

```

string[] formats = { "G", "F", "D", "X" };
ArrivalStatus status = ArrivalStatus.Late;
foreach (var fmt in formats)
    Console.WriteLine(status.ToString(fmt));

// The example displays the following output:
//      Late

```

```
//      Late
//      -1
//      FFFFFFFF
```

Iterate enumeration members

The [Enum](#) type does not implement the [IEnumerable](#) or [IEnumerable<T>](#) interface, which would enable you to iterate members of a collection by using a `foreach` (in C#), `for..in` (in F#), or `For Each` (in Visual Basic) construct. However, you can enumerate members in either of two ways.

- You can call the [GetNames](#) method to retrieve a string array containing the names of the enumeration members. Next, for each element of the string array, you can call the [Parse](#) method to convert the string to its equivalent enumeration value.
- The following example illustrates this approach.

C#

```
string[] names = Enum.GetNames(typeof(ArrivalStatus));
Console.WriteLine("Members of {0}:", typeof(ArrivalStatus).Name);
Array.Sort(names);
foreach (var name in names)
{
    ArrivalStatus status =
(ArrivalStatus)Enum.Parse(typeof(ArrivalStatus), name);
    Console.WriteLine("    {0} ({0:D})", status);
}
// The example displays the following output:
//      Members of ArrivalStatus:
//          Early (1)
//          Late (-1)
//          OnTime (0)
//          Unknown (-3)
```

- You can call the [GetValues](#) method to retrieve an array that contains the underlying values in the enumeration. Next, for each element of the array, you can call the [ToObject](#) method to convert the integer to its equivalent enumeration value. The following example illustrates this approach.

C#

```
var values = Enum.GetValues(typeof(ArrivalStatus));
Console.WriteLine("Members of {0}:", typeof(ArrivalStatus).Name);
foreach (ArrivalStatus status in values)
{
    Console.WriteLine("    {0} ({0:D})", status);
}
```

```
// The example displays the following output:  
//      Members of ArrivalStatus:  
//          OnTime (0)  
//          Early (1)  
//          Unknown (-3)  
//          Late (-1)
```

Non-exclusive members and the Flags attribute

One common use of an enumeration is to represent a set of mutually exclusive values. For example, an `ArrivalStatus` instance can have a value of `Early`, `OnTime`, or `Late`. It makes no sense for the value of an `ArrivalStatus` instance to reflect more than one enumeration constant.

In other cases, however, the value of an enumeration object can include multiple enumeration members, and each member represents a bit field in the enumeration value. The [FlagsAttribute](#) attribute can be used to indicate that the enumeration consists of bit fields. For example, an enumeration named `Pets` might be used to indicate the kinds of pets in a household. It can be defined as follows.

```
C#  
  
[Flags]  
public enum Pets  
{  
    None = 0, Dog = 1, Cat = 2, Bird = 4, Rodent = 8,  
    Reptile = 16, Other = 32  
};
```

The `Pets` enumeration can then be used as shown in the following example.

```
C#  
  
Pets familyPets = Pets.Dog | Pets.Cat;  
Console.WriteLine("Pets: {0:G} ({0:D})", familyPets);  
// The example displays the following output:  
//      Pets: Dog, Cat (3)
```

The following best practices should be used when defining a bitwise enumeration and applying the [FlagsAttribute](#) attribute.

- Use the [FlagsAttribute](#) custom attribute for an enumeration only if a bitwise operation (AND, OR, EXCLUSIVE OR) is to be performed on a numeric value.

- Define enumeration constants in powers of two, that is, 1, 2, 4, 8, and so on. This means the individual flags in combined enumeration constants do not overlap.
- Consider creating an enumerated constant for commonly used flag combinations. For example, if you have an enumeration used for file I/O operations that contains the enumerated constants `Read = 1` and `Write = 2`, consider creating the enumerated constant `ReadWrite = Read OR Write`, which combines the `Read` and `Write` flags. In addition, the bitwise OR operation used to combine the flags might be considered an advanced concept in some circumstances that should not be required for simple tasks.
- Use caution if you define a negative number as a flag enumerated constant because many flag positions might be set to 1, which might make your code confusing and encourage coding errors.
- A convenient way to test whether a flag is set in a numeric value is to call the instance `HasFlag` method, as shown in the following example.

C#

```
Pets familyPets = Pets.Dog | Pets.Cat;
if (familyPets.HasFlag(Pets.Dog))
    Console.WriteLine("The family has a dog.");
// The example displays the following output:
//      The family has a dog.
```

It is equivalent to performing a bitwise AND operation between the numeric value and the flag enumerated constant, which sets all bits in the numeric value to zero that do not correspond to the flag, and then testing whether the result of that operation is equal to the flag enumerated constant. This is illustrated in the following example.

C#

```
Pets familyPets = Pets.Dog | Pets.Cat;
if ((familyPets & Pets.Dog) == Pets.Dog)
    Console.WriteLine("The family has a dog.");
// The example displays the following output:
//      The family has a dog.
```

- Use `None` as the name of the flag enumerated constant whose value is zero. You cannot use the `None` enumerated constant in a bitwise AND operation to test for a flag because the result is always zero. However, you can perform a logical, not a bitwise, comparison between the numeric value and the `None` enumerated

constant to determine whether any bits in the numeric value are set. This is illustrated in the following example.

```
C#  
  
Pets familyPets = Pets.Dog | Pets.Cat;  
if (familyPets == Pets.None)  
    Console.WriteLine("The family has no pets.");  
else  
    Console.WriteLine("The family has pets.");  
// The example displays the following output:  
//     The family has pets.
```

- Do not define an enumeration value solely to mirror the state of the enumeration itself. For example, do not define an enumerated constant that merely marks the end of the enumeration. If you need to determine the last value of the enumeration, check for that value explicitly. In addition, you can perform a range check for the first and last enumerated constant if all values within the range are valid.

Add enumeration methods

Because enumeration types are defined by language structures, such as `enum` (C#), and `Enum` (Visual Basic), you cannot define custom methods for an enumeration type other than those methods inherited from the `Enum` class. However, you can use extension methods to add functionality to a particular enumeration type.

In the following example, the `Grades` enumeration represents the possible letter grades that a student may receive in a class. An extension method named `Passing` is added to the `Grades` type so that each instance of that type now "knows" whether it represents a passing grade or not. The `Extensions` class also contains a static read-write variable that defines the minimum passing grade. The return value of the `Passing` extension method reflects the current value of that variable.

```
C#  
  
using System;  
  
// Define an enumeration to represent student grades.  
public enum Grades { F = 0, D = 1, C = 2, B = 3, A = 4 };  
  
// Define an extension method for the Grades enumeration.  
public static class Extensions  
{  
    public static Grades minPassing = Grades.D;
```

```

    public static bool Passing(this Grades grade)
    {
        return grade >= minPassing;
    }
}

class Example8
{
    static void Main()
    {
        Grades g1 = Grades.D;
        Grades g2 = Grades.F;
        Console.WriteLine("{0} {1} a passing grade.", g1, g1.Passing() ? "is" : "is not");
        Console.WriteLine("{0} {1} a passing grade.", g2, g2.Passing() ? "is" : "is not");

        Extensions.minPassing = Grades.C;
        Console.WriteLine("\nRaising the bar!\n");
        Console.WriteLine("{0} {1} a passing grade.", g1, g1.Passing() ? "is" : "is not");
        Console.WriteLine("{0} {1} a passing grade.", g2, g2.Passing() ? "is" : "is not");
    }
}

// The example displays the following output:
//      D is a passing grade.
//      F is not a passing grade.
//
//      Raising the bar!
//
//      D is not a passing grade.
//      F is not a passing grade.

```

Examples

The following example demonstrates using an enumeration to represent named values and another enumeration to represent named bit fields.

C#

```

using System;

public class EnumTest {
    enum Days { Saturday, Sunday, Monday, Tuesday, Wednesday, Thursday,
Friday };
    enum BoilingPoints { Celsius = 100, Fahrenheit = 212 };
    [Flags]
    enum Colors { Red = 1, Green = 2, Blue = 4, Yellow = 8 };
}

```

```
public static void Main() {  
  
    Type weekdays = typeof(Days);  
    Type boiling = typeof(BoilingPoints);  
  
    Console.WriteLine("The days of the week, and their corresponding  
values in the Days Enum are:");  
  
    foreach ( string s in Enum.GetNames(weekdays) )  
        Console.WriteLine( "{0,-11}={1}", s, Enum.Format( weekdays,  
Enum.Parse(weekdays, s), "d"));  
  
    Console.WriteLine();  
    Console.WriteLine("Enums can also be created which have values that  
represent some meaningful amount.");  
    Console.WriteLine("The BoilingPoints Enum defines the following  
items, and corresponding values:");  
  
    foreach ( string s in Enum.GetNames(boiling) )  
        Console.WriteLine( "{0,-11}={1}", s, Enum.Format(boiling,  
Enum.Parse(boiling, s), "d"));  
  
    Colors myColors = Colors.Red | Colors.Blue | Colors.Yellow;  
    Console.WriteLine();  
    Console.WriteLine("myColors holds a combination of colors. Namely:  
{0}", myColors);  
}  
}
```

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



.NET feedback

.NET is an open source project.
Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

System.FlagsAttribute class

Article • 01/08/2024

This article provides supplementary remarks to the reference documentation for this API.

The [FlagsAttribute](#) attribute indicates that an enumeration can be treated as a bit field; that is, a set of flags.

Bit fields are generally used for lists of elements that might occur in combination, whereas enumeration constants are generally used for lists of mutually exclusive elements. Therefore, bit fields are designed to be combined with a bitwise `OR` operation to generate unnamed values, whereas enumerated constants are not. Languages vary in their use of bit fields compared to enumeration constants.

Attributes of the FlagsAttribute

[AttributeUsageAttribute](#) is applied to this class, and its [Inherited](#) property specifies `false`. This attribute can only be applied to enumerations.

Guidelines for FlagsAttribute and enum

- Use the [FlagsAttribute](#) custom attribute for an enumeration only if a bitwise operation (AND, OR, EXCLUSIVE OR) is to be performed on a numeric value.
- Define enumeration constants in powers of two, that is, 1, 2, 4, 8, and so on. This means the individual flags in combined enumeration constants do not overlap.
- Consider creating an enumerated constant for commonly used flag combinations. For example, if you have an enumeration used for file I/O operations that contains the enumerated constants `Read = 1` and `Write = 2`, consider creating the enumerated constant `ReadWrite = Read OR Write`, which combines the `Read` and `Write` flags. In addition, the bitwise OR operation used to combine the flags might be considered an advanced concept in some circumstances that should not be required for simple tasks.
- Use caution if you define a negative number as a flag enumerated constant because many flag positions might be set to 1, which might make your code confusing and encourage coding errors.

- A convenient way to test whether a flag is set in a numeric value is to perform a bitwise AND operation between the numeric value and the flag enumerated constant, which sets all bits in the numeric value to zero that do not correspond to the flag, then test whether the result of that operation is equal to the flag enumerated constant.
- Use `None` as the name of the flag enumerated constant whose value is zero. You cannot use the `None` enumerated constant in a bitwise AND operation to test for a flag because the result is always zero. However, you can perform a logical, not a bitwise, comparison between the numeric value and the `None` enumerated constant to determine whether any bits in the numeric value are set.

If you create a value enumeration instead of a flags enumeration, it is still worthwhile to create a `None` enumerated constant. The reason is that by default the memory used for the enumeration is initialized to zero by the common language runtime. Consequently, if you do not define a constant whose value is zero, the enumeration will contain an illegal value when it is created.

If there is an obvious default case your application needs to represent, consider using an enumerated constant whose value is zero to represent the default. If there is no default case, consider using an enumerated constant whose value is zero that means the case that is not represented by any of the other enumerated constants.

- Do not define an enumeration value solely to mirror the state of the enumeration itself. For example, do not define an enumerated constant that merely marks the end of the enumeration. If you need to determine the last value of the enumeration, check for that value explicitly. In addition, you can perform a range check for the first and last enumerated constant if all values within the range are valid.
- Do not specify enumerated constants that are reserved for future use.
- When you define a method or property that takes an enumerated constant as a value, consider validating the value. The reason is that you can cast a numeric value to the enumeration type even if that numeric value is not defined in the enumeration.

Examples

The following example illustrates the use of the `FlagsAttribute` attribute and shows the effect on the `ToString` method of using `FlagsAttribute` on an `Enum` declaration.

C#

```
using System;

class Example
{
    // Define an Enum without FlagsAttribute.
    enum SingleHue : short
    {
        None = 0,
        Black = 1,
        Red = 2,
        Green = 4,
        Blue = 8
    };

    // Define an Enum with FlagsAttribute.
    [Flags]
    enum MultiHue : short
    {
        None = 0,
        Black = 1,
        Red = 2,
        Green = 4,
        Blue = 8
    };

    static void Main()
    {
        // Display all possible combinations of values.
        Console.WriteLine(
            "All possible combinations of values without FlagsAttribute:");
        for (int val = 0; val <= 16; val++)
            Console.WriteLine("{0,3} - {1:G}", val, (SingleHue)val);

        // Display all combinations of values, and invalid values.
        Console.WriteLine(
            "\nAll possible combinations of values with FlagsAttribute:");
        for (int val = 0; val <= 16; val++)
            Console.WriteLine("{0,3} - {1:G}", val, (MultiHue)val);
    }
}

// The example displays the following output:
//      All possible combinations of values without FlagsAttribute:
//          0 - None
//          1 - Black
//          2 - Red
//          3 - 3
//          4 - Green
//          5 - 5
//          6 - 6
//          7 - 7
//          8 - Blue
//          9 - 9
```

```

//      10 - 10
//      11 - 11
//      12 - 12
//      13 - 13
//      14 - 14
//      15 - 15
//      16 - 16
//
//      All possible combinations of values with FlagsAttribute:
//      0 - None
//      1 - Black
//      2 - Red
//      3 - Black, Red
//      4 - Green
//      5 - Black, Green
//      6 - Red, Green
//      7 - Black, Red, Green
//      8 - Blue
//      9 - Black, Blue
//      10 - Red, Blue
//      11 - Black, Red, Blue
//      12 - Green, Blue
//      13 - Black, Green, Blue
//      14 - Red, Green, Blue
//      15 - Black, Red, Green, Blue
//      16 - 16

```

The preceding example defines two color-related enumerations, `SingleHue` and `MultiHue`. The latter has the `FlagsAttribute` attribute; the former does not. The example shows the difference in behavior when a range of integers, including integers that do not represent underlying values of the enumeration type, are cast to the enumeration type and their string representations displayed. For example, note that 3 cannot be represented as a `SingleHue` value because 3 is not the underlying value of any `SingleHue` member, whereas the `FlagsAttribute` attribute makes it possible to represent 3 as a `MultiHue` value of `Black, Red`.

The following example defines another enumeration with the `FlagsAttribute` attribute and shows how to use bitwise logical and equality operators to determine whether one or more bit fields are set in an enumeration value. You can also use the `Enum.HasFlag` method to do that, but that is not shown in this example.

C#

```

using System;

[Flags]
public enum PhoneService
{
    None = 0,

```

```

        LandLine = 1,
        Cell = 2,
        Fax = 4,
        Internet = 8,
        Other = 16
    }

    public class Example1
    {
        public static void Main()
        {
            // Define three variables representing the types of phone service
            // in three households.
            var household1 = PhoneService.LandLine | PhoneService.Cell |
                PhoneService.Internet;
            var household2 = PhoneService.None;
            var household3 = PhoneService.Cell | PhoneService.Internet;

            // Store the variables in an array for ease of access.
            PhoneService[] households = { household1, household2, household3 };

            // Which households have no service?
            for (int ctr = 0; ctr < households.Length; ctr++)
                Console.WriteLine("Household {0} has phone service: {1}",
                    ctr + 1,
                    households[ctr] == PhoneService.None ?
                        "No" : "Yes");
            Console.WriteLine();

            // Which households have cell phone service?
            for (int ctr = 0; ctr < households.Length; ctr++)
                Console.WriteLine("Household {0} has cell phone service: {1}",
                    ctr + 1,
                    (households[ctr] & PhoneService.Cell) ==
PhoneService.Cell ?
                        "Yes" : "No");
            Console.WriteLine();

            // Which households have cell phones and land lines?
            var cellAndLand = PhoneService.Cell | PhoneService.LandLine;
            for (int ctr = 0; ctr < households.Length; ctr++)
                Console.WriteLine("Household {0} has cell and land line service:
{1}",
                    ctr + 1,
                    (households[ctr] & cellAndLand) == cellAndLand ?
                        "Yes" : "No");
            Console.WriteLine();

            // List all types of service of each household?//
            for (int ctr = 0; ctr < households.Length; ctr++)
                Console.WriteLine("Household {0} has: {1:G}",
                    ctr + 1, households[ctr]);
            Console.WriteLine();
        }
    }

```

```
}
```

```
// The example displays the following output:
```

```
// Household 1 has phone service: Yes
```

```
// Household 2 has phone service: No
```

```
// Household 3 has phone service: Yes
```

```
//
```

```
// Household 1 has cell phone service: Yes
```

```
// Household 2 has cell phone service: No
```

```
// Household 3 has cell phone service: Yes
```

```
//
```

```
// Household 1 has cell and land line service: Yes
```

```
// Household 2 has cell and land line service: No
```

```
// Household 3 has cell and land line service: No
```

```
//
```

```
// Household 1 has: LandLine, Cell, Internet
```

```
// Household 2 has: None
```

```
// Household 3 has: Cell, Internet
```

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

.NET is an open source project.
Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

Handle and raise events

Article • 10/04/2022

Events in .NET are based on the delegate model. The delegate model follows the [observer design pattern](#), which enables a subscriber to register with and receive notifications from a provider. An event sender pushes a notification that an event has happened, and an event receiver receives that notification and defines a response to it. This article describes the major components of the delegate model, how to consume events in applications, and how to implement events in your code.

Events

An event is a message sent by an object to signal the occurrence of an action. The action can be caused by user interaction, such as a button click, or it can result from some other program logic, such as changing a property's value. The object that raises the event is called the *event sender*. The event sender doesn't know which object or method will receive (handle) the events it raises. The event is typically a member of the event sender; for example, the [Click](#) event is a member of the [Button](#) class, and the [PropertyChanged](#) event is a member of the class that implements the [INotifyPropertyChanged](#) interface.

To define an event, you use the C# [event](#) or the Visual Basic [Event](#) keyword in the signature of your event class, and specify the type of delegate for the event. Delegates are described in the next section.

Typically, to raise an event, you add a method that is marked as `protected` and `virtual` (in C#) or `Protected` and `Overridable` (in Visual Basic). Name this method `On EventName`; for example, `OnDataReceived`. The method should take one parameter that specifies an event data object, which is an object of type [EventArgs](#) or a derived type. You provide this method to enable derived classes to override the logic for raising the event. A derived class should always call the `On EventName` method of the base class to ensure that registered delegates receive the event.

The following example shows how to declare an event named `ThresholdReached`. The event is associated with the [EventHandler](#) delegate and raised in a method named `OnThresholdReached`.

C#

```
class Counter  
{
```

```
public event EventHandler ThresholdReached;

protected virtual void OnThresholdReached(EventArgs e)
{
    ThresholdReached?.Invoke(this, e);
}

// provide remaining implementation for the class
}
```

Delegates

A delegate is a type that holds a reference to a method. A delegate is declared with a signature that shows the return type and parameters for the methods it references, and it can hold references only to methods that match its signature. A delegate is thus equivalent to a type-safe function pointer or a callback. A delegate declaration is sufficient to define a delegate class.

Delegates have many uses in .NET. In the context of events, a delegate is an intermediary (or pointer-like mechanism) between the event source and the code that handles the event. You associate a delegate with an event by including the delegate type in the event declaration, as shown in the example in the previous section. For more information about delegates, see the [Delegate](#) class.

.NET provides the [EventHandler](#) and [EventHandler<TEventArgs>](#) delegates to support most event scenarios. Use the [EventHandler](#) delegate for all events that don't include event data. Use the [EventHandler<TEventArgs>](#) delegate for events that include data about the event. These delegates have no return type value and take two parameters (an object for the source of the event and an object for event data).

Delegates are [multicast](#), which means that they can hold references to more than one event-handling method. For more information, see the [Delegate](#) reference page. Delegates provide flexibility and fine-grained control in event handling. A delegate acts as an event dispatcher for the class that raises the event by maintaining a list of registered event handlers for the event.

For scenarios where the [EventHandler](#) and [EventHandler<TEventArgs>](#) delegates don't work, you can define a delegate. Scenarios that require you to define a delegate are rare, such as when you must work with code that doesn't recognize generics. You mark a delegate with the C# [delegate](#) and Visual Basic [Delegate](#) keyword in the declaration. The following example shows how to declare a delegate named

ThresholdReachedEventHandler:

C#

```
public delegate void ThresholdReachedEventHandler(object sender,  
ThresholdReachedEventArgs e);
```

Event data

Data that is associated with an event can be provided through an event data class. .NET provides many event data classes that you can use in your applications. For example, the [SerialDataReceivedEventArgs](#) class is the event data class for the [SerialPort.DataReceived](#) event. .NET follows a naming pattern of ending all event data classes with `EventArgs`. You determine which event data class is associated with an event by looking at the delegate for the event. For example, the [SerialDataReceivedEventHandler](#) delegate includes the [SerialDataReceivedEventArgs](#) class as one of its parameters.

The [EventArgs](#) class is the base type for all event data classes. [EventArgs](#) is also the class you use when an event doesn't have any data associated with it. When you create an event that is only meant to notify other classes that something happened and doesn't need to pass any data, include the [EventArgs](#) class as the second parameter in the delegate. You can pass the [EventArgs.Empty](#) value when no data is provided. The [EventHandler](#) delegate includes the [EventArgs](#) class as a parameter.

When you want to create a customized event data class, create a class that derives from [EventArgs](#), and then provide any members needed to pass data that is related to the event. Typically, you should use the same naming pattern as .NET and end your event data class name with `EventArgs`.

The following example shows an event data class named `ThresholdReachedEventArgs`. It contains properties that are specific to the event being raised:

C#

```
public class ThresholdReachedEventArgs : EventArgs  
{  
    public int Threshold { get; set; }  
    public DateTime TimeReached { get; set; }  
}
```

Event handlers

To respond to an event, you define an event handler method in the event receiver. This method must match the signature of the delegate for the event you're handling. In the

event handler, you perform the actions that are required when the event is raised, such as collecting user input after the user clicks a button. To receive notifications when the event occurs, your event handler method must subscribe to the event.

The following example shows an event handler method named `c_ThresholdReached` that matches the signature for the `EventHandler` delegate. The method subscribes to the `ThresholdReached` event.

```
C#  
  
class ProgramTwo  
{  
    static void Main()  
    {  
        var c = new Counter();  
        c.ThresholdReached += c_ThresholdReached;  
  
        // provide remaining implementation for the class  
    }  
  
    static void c_ThresholdReached(object sender, EventArgs e)  
    {  
        Console.WriteLine("The threshold was reached.");  
    }  
}
```

Static and dynamic event handlers

.NET allows subscribers to register for event notifications either statically or dynamically. Static event handlers are in effect for the entire life of the class whose events they handle. Dynamic event handlers are explicitly activated and deactivated during program execution, usually in response to some conditional program logic. For example, they can be used if event notifications are needed only under certain conditions or if an application provides multiple event handlers and run-time conditions define the appropriate one to use. The example in the previous section shows how to dynamically add an event handler. For more information, see [Events](#) (in Visual Basic) and [Events](#) (in C#).

Raising multiple events

If your class raises multiple events, the compiler generates one field per event delegate instance. If the number of events is large, the storage cost of one field per delegate might not be acceptable. For those situations, .NET provides event properties that you can use with another data structure of your choice to store event delegates.

Event properties consist of event declarations accompanied by event accessors. Event accessors are methods that you define to add or remove event delegate instances from the storage data structure.

Note

The event properties are slower than the event fields because each event delegate must be retrieved before it can be invoked.

The trade-off is between memory and speed. If your class defines many events that are infrequently raised, you'll want to implement event properties. For more information, see [How to: Handle Multiple Events Using Event Properties](#).

Related articles

| Title | Description |
|---|--|
| How to: Raise and Consume Events | Contains examples of raising and consuming events. |
| How to: Handle Multiple Events Using Event Properties | Shows how to use event properties to handle multiple events. |
| Observer Design Pattern | Describes the design pattern that enables a subscriber to register with and receive notifications from a provider. |

See also

- [EventHandler](#)
- [EventHandler<TEventArgs>](#)
- [EventArgs](#)
- [Delegate](#)
- [Events \(Visual Basic\)](#)
- [Events \(C# Programming Guide\)](#)
- [Events and routed events overview \(UWP apps\)](#)
- [Events in Windows Store 8.x apps](#)

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

How to: Raise and Consume Events

Article • 10/04/2022

The examples in this article show how to work with events. They include examples of the `EventHandler` delegate, the `EventHandler<TEventArgs>` delegate, and a custom delegate to illustrate events with and without data.

The examples use concepts described in the [Events](#) article.

Example 1

The first example shows how to raise and consume an event that doesn't have data. It contains a class named `Counter` that has an event called `ThresholdReached`. This event is raised when a counter value equals or exceeds a threshold value. The `EventHandler` delegate is associated with the event because no event data is provided.

C#

```
using System;

namespace ConsoleApplication1
{
    class ProgramOne
    {
        static void Main(string[] args)
        {
            Counter c = new Counter(new Random().Next(10));
            c.ThresholdReached += c_ThresholdReached;

            Console.WriteLine("press 'a' key to increase total");
            while (Console.ReadKey(true).KeyChar == 'a')
            {
                Console.WriteLine("adding one");
                c.Add(1);
            }
        }

        static void c_ThresholdReached(object sender, EventArgs e)
        {
            Console.WriteLine("The threshold was reached.");
            Environment.Exit(0);
        }
    }

    class Counter
    {
        private int threshold;
        private int total;
```

```

public Counter(int passedThreshold)
{
    threshold = passedThreshold;
}

public void Add(int x)
{
    total += x;
    if (total >= threshold)
    {
        ThresholdReached?.Invoke(this, EventArgs.Empty);
    }
}

public event EventHandler ThresholdReached;
}
}

```

Example 2

The second example shows how to raise and consume an event that provides data. The `EventHandler<TEventArgs>` delegate is associated with the event, and an instance of a custom event data object is provided.

C#

```

using System;

namespace ConsoleApplication3
{
    class ProgramThree
    {
        static void Main(string[] args)
        {
            Counter c = new Counter(new Random().Next(10));
            c.ThresholdReached += c_ThresholdReached;

            Console.WriteLine("press 'a' key to increase total");
            while (Console.ReadKey(true).KeyChar == 'a')
            {
                Console.WriteLine("adding one");
                c.Add(1);
            }
        }

        static void c_ThresholdReached(object sender,
ThresholdReachedEventArgs e)
        {
            Console.WriteLine("The threshold of {0} was reached at {1}.",
e.Threshold, e.TimeReached);
        }
    }
}

```

```

        Environment.Exit(0);
    }
}

class Counter
{
    private int threshold;
    private int total;

    public Counter(int passedThreshold)
    {
        threshold = passedThreshold;
    }

    public void Add(int x)
    {
        total += x;
        if (total >= threshold)
        {
            ThresholdReachedEventArgs args = new
ThresholdReachedEventArgs();
            args.Threshold = threshold;
            args.TimeReached = DateTime.Now;
            OnThresholdReached(args);
        }
    }

    protected virtual void OnThresholdReached(ThresholdReachedEventArgs
e)
    {
        EventHandler<ThresholdReachedEventArgs> handler =
ThresholdReached;
        if (handler != null)
        {
            handler(this, e);
        }
    }

    public event EventHandler<ThresholdReachedEventArgs>
ThresholdReached;
}

public class ThresholdReachedEventArgs : EventArgs
{
    public int Threshold { get; set; }
    public DateTime TimeReached { get; set; }
}
}

```

Example 3

The third example shows how to declare a delegate for an event. The delegate is named `ThresholdReachedEventHandler`. This example is just an illustration. Typically, you don't have to declare a delegate for an event because you can use either the `EventHandler` or the `EventHandler<TEventArgs>` delegate. You should declare a delegate only in rare scenarios, such as making your class available to legacy code that can't use generics.

C#

```
using System;

namespace ConsoleApplication4
{
    class ProgramFour
    {
        static void Main(string[] args)
        {
            Counter c = new Counter(new Random().Next(10));
            c.ThresholdReached += c_ThresholdReached;

            Console.WriteLine("press 'a' key to increase total");
            while (Console.ReadKey(true).KeyChar == 'a')
            {
                Console.WriteLine("adding one");
                c.Add(1);
            }
        }

        static void c_ThresholdReached(Object sender,
ThresholdReachedEventArgs e)
        {
            Console.WriteLine("The threshold of {0} was reached at {1}.",
e.Threshold, e.TimeReached);
            Environment.Exit(0);
        }
    }

    class Counter
    {
        private int threshold;
        private int total;

        public Counter(int passedThreshold)
        {
            threshold = passedThreshold;
        }

        public void Add(int x)
        {
            total += x;
            if (total >= threshold)
            {
                ThresholdReachedEventArgs args = new
```

```
ThresholdReachedEventArgs();
    args.Threshold = threshold;
    args.TimeReached = DateTime.Now;
    OnThresholdReached(args);
}
}

protected virtual void OnThresholdReached(ThresholdReachedEventArgs e)
{
    ThresholdReachedEventHandler handler = ThresholdReached;
    if (handler != null)
    {
        handler(this, e);
    }
}

public event ThresholdReachedEventHandler ThresholdReached;
}

public class ThresholdReachedEventArgs : EventArgs
{
    public int Threshold { get; set; }
    public DateTime TimeReached { get; set; }
}

public delegate void ThresholdReachedEventHandler(Object sender,
ThresholdReachedEventArgs e);
}
```

See also

- [Events](#)

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



.NET feedback

The .NET documentation is open source. Provide feedback [here](#).

 [Open a documentation issue](#)

 [Provide product feedback](#)

How to: Handle Multiple Events Using Event Properties

Article • 09/15/2021

To use event properties, you define the event properties in the class that raises the events, and then set the delegates for the event properties in classes that handle the events. To implement multiple event properties in a class, the class should internally store and maintain the delegate defined for each event. For each field-like-event, a corresponding backing-field reference type is generated. This can lead to unnecessary allocations when the number of events increases. As an alternative, a common approach is to maintain an [EventHandlerList](#) which stores events by key.

To store the delegates for each event, you can use the [EventHandlerList](#) class, or implement your own collection. The collection class must provide methods for setting, accessing, and retrieving the event handler delegate based on the event key. For example, you could use a [Hashtable](#) class, or derive a custom class from the [DictionaryBase](#) class. The implementation details of the delegate collection do not need to be exposed outside your class.

Each event property within the class defines an add accessor method and a remove accessor method. The add accessor for an event property adds the input delegate instance to the delegate collection. The remove accessor for an event property removes the input delegate instance from the delegate collection. The event property accessors use the predefined key for the event property to add and remove instances from the delegate collection.

To handle multiple events using event properties

1. Define a delegate collection within the class that raises the events.
2. Define a key for each event.
3. Define the event properties in the class that raises the events.
4. Use the delegate collection to implement the add and remove accessor methods for the event properties.
5. Use the public event properties to add and remove event handler delegates in the classes that handle the events.

Example

The following C# example implements the event properties `MouseDown` and `MouseUp`, using an `EventHandlerList` to store each event's delegate. The keywords of the event property constructs are in bold type.

C#

```
// The class SampleControl defines two event properties, MouseUp and
MouseDown.
class SampleControl : Component
{
    // :
    // Define other control methods and properties.
    //

    // Define the delegate collection.
    protected EventHandlerList listEventDelegates = new EventHandlerList();

    // Define a unique key for each event.
    static readonly object mouseDownEventKey = new object();
    static readonly object mouseUpEventKey = new object();

    // Define the MouseDown event property.
    public event MouseEventHandler MouseDown
    {
        // Add the input delegate to the collection.
        add
        {
            listEventDelegates.AddHandler(mouseDownEventKey, value);
        }
        // Remove the input delegate from the collection.
        remove
        {
            listEventDelegates.RemoveHandler(mouseDownEventKey, value);
        }
    }

    // Raise the event with the delegate specified by mouseDownEventKey
    private void OnMouseDown(MouseEventArgs e)
    {
        MouseEventHandler mouseEventDelegate =
            (MouseEventHandler)listEventDelegates[mouseDownEventKey];
        mouseEventDelegate(this, e);
    }

    // Define the MouseUp event property.
    public event MouseEventHandler MouseUp
    {
        // Add the input delegate to the collection.
        add
        {
```

```
        listEventDelegates.AddHandler(mouseUpEventKey, value);
    }
    // Remove the input delegate from the collection.
    remove
    {
        listEventDelegates.RemoveHandler(mouseUpEventKey, value);
    }
}

// Raise the event with the delegate specified by mouseUpEventKey
private void OnMouseUp(MouseEventArgs e)
{
    MouseEventHandler mouseEventDelegate =
        (MouseEventHandler)listEventDelegates[mouseUpEventKey];
    mouseEventDelegate(this, e);
}
}
```

See also

- [System.ComponentModel.EventHandlerList](#)
- [Events](#)
- [Control.Events](#)
- [How to: Declare Custom Events To Conserve Memory](#)

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback [here](#).

 [Open a documentation issue](#)

 [Provide product feedback](#)

Observer design pattern

Article • 05/25/2023

The observer design pattern enables a subscriber to register with and receive notifications from a provider. It's suitable for any scenario that requires push-based notification. The pattern defines a *provider* (also known as a *subject* or an *observable*) and zero, one, or more *observers*. Observers register with the provider, and whenever a predefined condition, event, or state change occurs, the provider automatically notifies all observers by invoking a delegate. In this method call, the provider can also provide current state information to observers. In .NET, the observer design pattern is applied by implementing the generic [System.IObservable<T>](#) and [System.IObserver<T>](#) interfaces. The generic type parameter represents the type that provides notification information.

When to apply the pattern

The observer design pattern is suitable for distributed push-based notifications, because it supports a clean separation between two different components or application layers, such as a data source (business logic) layer and a user interface (display) layer. The pattern can be implemented whenever a provider uses callbacks to supply its clients with current information.

Implementing the pattern requires that you provide the following details:

- A provider or subject, which is the object that sends notifications to observers. A provider is a class or structure that implements the [IObservable<T>](#) interface. The provider must implement a single method, [IObservable<T>.Subscribe](#), which is called by observers that wish to receive notifications from the provider.
- An observer, which is an object that receives notifications from a provider. An observer is a class or structure that implements the [IObserver<T>](#) interface. The observer must implement three methods, all of which are called by the provider:
 - [IObserver<T>.OnNext](#), which supplies the observer with new or current information.
 - [IObserver<T>.OnError](#), which informs the observer that an error has occurred.
 - [IObserver<T>.OnCompleted](#), which indicates that the provider has finished sending notifications.
- A mechanism that allows the provider to keep track of observers. Typically, the provider uses a container object, such as a [System.Collections.Generic.List<T>](#) object, to hold references to the [IObserver<T>](#) implementations that have subscribed to notifications. Using a storage container for this purpose enables the

provider to handle zero to an unlimited number of observers. The order in which observers receive notifications isn't defined; the provider is free to use any method to determine the order.

- An [IDisposable](#) implementation that enables the provider to remove observers when notification is complete. Observers receive a reference to the [IDisposable](#) implementation from the [Subscribe](#) method, so they can also call the [IDisposable.Dispose](#) method to unsubscribe before the provider has finished sending notifications.
- An object that contains the data that the provider sends to its observers. The type of this object corresponds to the generic type parameter of the [IObservable<T>](#) and [IObserver<T>](#) interfaces. Although this object can be the same as the [IObservable<T>](#) implementation, most commonly it's a separate type.

ⓘ Note

In addition to implementing the observer design pattern, you may be interested in exploring libraries that are built using the [IObservable<T>](#) and [IObserver<T>](#) interfaces. For example, [Reactive Extensions for .NET \(Rx\)](#) consist of a set of extension methods and LINQ standard sequence operators to support asynchronous programming.

Implement the pattern

The following example uses the observer design pattern to implement an airport baggage claim information system. A [BaggageInfo](#) class provides information about arriving flights and the carousels where baggage from each flight is available for pickup. It's shown in the following example.

C#

```
namespace Observables.Example;

public readonly record struct BaggageInfo(
    int FlightNumber,
    string From,
    int Carousel);
```

A [BaggageHandler](#) class is responsible for receiving information about arriving flights and baggage claim carousels. Internally, it maintains two collections:

- `_observers`: A collection of clients that observe updated information.
- `_flights`: A collection of flights and their assigned carousels.

The source code for the `BaggageHandler` class is shown in the following example.

C#

```
namespace Observables.Example;

public sealed class BaggageHandler : IObservable<BaggageInfo>
{
    private readonly HashSet<IObserver<BaggageInfo>> _observers = new();
    private readonly HashSet<BaggageInfo> _flights = new();

    public IDisposable Subscribe(IObserver<BaggageInfo> observer)
    {
        // Check whether observer is already registered. If not, add it.
        if (_observers.Add(observer))
        {
            // Provide observer with existing data.
            foreach (BaggageInfo item in _flights)
            {
                observer.OnNext(item);
            }
        }

        return new Unsubscriber<BaggageInfo>(_observers, observer);
    }

    // Called to indicate all baggage is now unloaded.
    public void BaggageStatus(int flightNumber) =>
        BaggageStatus(flightNumber, string.Empty, 0);

    public void BaggageStatus(int flightNumber, string from, int carousel)
    {
        var info = new BaggageInfo(flightNumber, from, carousel);

        // Carousel is assigned, so add new info object to list.
        if (carousel > 0 && _flights.Add(info))
        {
            foreach (IObserver<BaggageInfo> observer in _observers)
            {
                observer.OnNext(info);
            }
        }
        else if (carousel is 0)
        {
            // Baggage claim for flight is done.
            if (_flights.RemoveWhere(
                flight => flight.FlightNumber == info.FlightNumber) > 0)
            {
                foreach (IObserver<BaggageInfo> observer in _observers)
                {

```

```

        observer.OnNext(info);
    }
}
}

public void LastBaggageClaimed()
{
    foreach (I0server<BaggageInfo> observer in _observers)
    {
        observer.OnCompleted();
    }

    _observers.Clear();
}
}

```

Clients that wish to receive updated information call the `BaggageHandler.Subscribe` method. If the client hasn't previously subscribed to notifications, a reference to the client's `I0server<T>` implementation is added to the `_observers` collection.

The overloaded `BaggageHandler.BaggageStatus` method can be called to indicate that baggage from a flight either is being unloaded or is no longer being unloaded. In the first case, the method is passed a flight number, the airport from which the flight originated, and the carousel where baggage is being unloaded. In the second case, the method is passed only a flight number. For baggage that is being unloaded, the method checks whether the `BaggageInfo` information passed to the method exists in the `_flights` collection. If it doesn't, the method adds the information and calls each observer's `OnNext` method. For flights whose baggage is no longer being unloaded, the method checks whether information on that flight is stored in the `_flights` collection. If it is, the method calls each observer's `OnNext` method and removes the `BaggageInfo` object from the `_flights` collection.

When the last flight of the day has landed and its baggage has been processed, the `BaggageHandler.LastBaggageClaimed` method is called. This method calls each observer's `OnCompleted` method to indicate that all notifications have completed, and then clears the `_observers` collection.

The provider's `Subscribe` method returns an `IDisposable` implementation that enables observers to stop receiving notifications before the `OnCompleted` method is called. The source code for this `Unsubscriber<Of BaggageInfo>` class is shown in the following example. When the class is instantiated in the `BaggageHandler.Subscribe` method, it's passed a reference to the `_observers` collection and a reference to the observer that is added to the collection. These references are assigned to local variables. When the

object's `Dispose` method is called, it checks whether the observer still exists in the `_observers` collection, and, if it does, removes the observer.

C#

```
namespace Observables.Example;

internal sealed class Unsubscriber<BaggageInfo> : IDisposable
{
    private readonly ISet<IObserver<BaggageInfo>> _observers;
    private readonly IObserver<BaggageInfo> _observer;

    internal Unsubscriber(
        ISet<IObserver<BaggageInfo>> observers,
        IObserver<BaggageInfo> observer) => (_observers, _observer) =
(observers, observer);

    public void Dispose() => _observers.Remove(_observer);
}
```

The following example provides an `IObserver<T>` implementation named `ArrivalsMonitor`, which is a base class that displays baggage claim information. The information is displayed alphabetically, by the name of the originating city. The methods of `ArrivalsMonitor` are marked as `overridable` (in Visual Basic) or `virtual` (in C#), so they can be overridden in a derived class.

C#

```
namespace Observables.Example;

public class ArrivalsMonitor : IObserver<BaggageInfo>
{
    private readonly string _name;
    private readonly List<string> _flights = new();
    private readonly string _format = "{0,-20} {1,5} {2, 3}";
    private IDisposable? _cancellation;

    public ArrivalsMonitor(string name)
    {
        ArgumentException.ThrowIfNullOrEmpty(name);
        _name = name;
    }

    public virtual void Subscribe(BaggageHandler provider) =>
        _cancellation = provider.Subscribe(this);

    public virtual void Unsubscribe()
    {
        _cancellation?.Dispose();
        _flights.Clear();
    }
}
```

```
}

public virtual void OnCompleted() => _flights.Clear();

// No implementation needed: Method is not called by the BaggageHandler
class.
public virtual void OnError(Exception e)
{
    // No implementation.
}

// Update information.
public virtual void OnNext(BaggageInfo info)
{
    bool updated = false;

    // Flight has unloaded its baggage; remove from the monitor.
    if (info.Carousel is 0)
    {
        string flightNumber = string.Format("{0,5}", info.FlightNumber);
        for (int index = _flights.Count - 1; index >= 0; index--)
        {
            string flightInfo = _flights[index];
            if (flightInfo.Substring(21, 5).Equals(flightNumber))
            {
                updated = true;
                _flights.RemoveAt(index);
            }
        }
    }
    else
    {
        // Add flight if it doesn't exist in the collection.
        string flightInfo = string.Format(_format, info.From,
info.FlightNumber, info.Carousel);
        if (_flights.Contains(flightInfo) is false)
        {
            _flights.Add(flightInfo);
            updated = true;
        }
    }

    if (updated)
    {
        _flights.Sort();
        Console.WriteLine($"Arrivals information from {_name}");
        foreach (string flightInfo in _flights)
        {
            Console.WriteLine(flightInfo);
        }

        Console.WriteLine();
    }
}
}
```

The `ArrivalsMonitor` class includes the `Subscribe` and `Unsubscribe` methods. The `Subscribe` method enables the class to save the `IDisposable` implementation returned by the call to `Subscribe` to a private variable. The `Unsubscribe` method enables the class to unsubscribe from notifications by calling the provider's `Dispose` implementation. `ArrivalsMonitor` also provides implementations of the `OnNext`, `OnError`, and `OnCompleted` methods. Only the `OnNext` implementation contains a significant amount of code. The method works with a private, sorted, generic `List<T>` object that maintains information about the airports of origin for arriving flights and the carousels on which their baggage is available. If the `BaggageHandler` class reports a new flight arrival, the `OnNext` method implementation adds information about that flight to the list. If the `BaggageHandler` class reports that the flight's baggage has been unloaded, the `OnNext` method removes that flight from the list. Whenever a change is made, the list is sorted and displayed to the console.

The following example contains the application entry point that instantiates the `BaggageHandler` class and two instances of the `ArrivalsMonitor` class, and uses the `BaggageHandler.BaggageStatus` method to add and remove information about arriving flights. In each case, the observers receive updates and correctly display baggage claim information.

C#

```
using Observables.Example;

BaggageHandler provider = new();
ArrivalsMonitor observer1 = new("BaggageClaimMonitor1");
ArrivalsMonitor observer2 = new("SecurityExit");

provider.BaggageStatus(712, "Detroit", 3);
observer1.Subscribe(provider);

provider.BaggageStatus(712, "Kalamazoo", 3);
provider.BaggageStatus(400, "New York-Kennedy", 1);
provider.BaggageStatus(712, "Detroit", 3);
observer2.Subscribe(provider);

provider.BaggageStatus(511, "San Francisco", 2);
provider.BaggageStatus(712);
observer2.Unsubscribe();

provider.BaggageStatus(400);
provider.LastBaggageClaimed();

// Sample output:
//   Arrivals information from BaggageClaimMonitor1
//   Detroit          712      3
```

```

//  

// Arrivals information from BaggageClaimMonitor1  

// Detroit 712 3  

// Kalamazoo 712 3  

//  

// Arrivals information from BaggageClaimMonitor1  

// Detroit 712 3  

// Kalamazoo 712 3  

// New York-Kennedy 400 1  

//  

// Arrivals information from SecurityExit  

// Detroit 712 3  

//  

// Arrivals information from SecurityExit  

// Detroit 712 3  

// Kalamazoo 712 3  

//  

// Arrivals information from SecurityExit  

// Detroit 712 3  

// Kalamazoo 712 3  

// New York-Kennedy 400 1  

//  

// Arrivals information from BaggageClaimMonitor1  

// Detroit 712 3  

// Kalamazoo 712 3  

// New York-Kennedy 400 1  

// San Francisco 511 2  

//  

// Arrivals information from SecurityExit  

// Detroit 712 3  

// Kalamazoo 712 3  

// New York-Kennedy 400 1  

// San Francisco 511 2  

//  

// Arrivals information from BaggageClaimMonitor1  

// New York-Kennedy 400 1  

// San Francisco 511 2  

//  

// Arrivals information from SecurityExit  

// New York-Kennedy 400 1  

// San Francisco 511 2  

//  

// Arrivals information from BaggageClaimMonitor1  

// San Francisco 511 2

```

Related articles

| Title | Description |
|--|--|
| Observer Design Pattern Best Practices | Describes best practices to adopt when developing applications that implement the observer design pattern. |

| Title | Description |
|-------------------------------|---|
| How to: Implement a Provider | Provides a step-by-step implementation of a provider for a temperature monitoring application. |
| How to: Implement an Observer | Provides a step-by-step implementation of an observer for a temperature monitoring application. |

 Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

Observer Design Pattern Best Practices

Article • 09/15/2021

In .NET, the observer design pattern is implemented as a set of interfaces. The [System.IObservable<T>](#) interface represents the data provider, which is also responsible for providing an [IDisposable](#) implementation that lets observers unsubscribe from notifications. The [System.IObserver<T>](#) interface represents the observer. This topic describes the best practices that developers should follow when implementing the observer design pattern using these interfaces.

Threading

Typically, a provider implements the [IObservable<T>.Subscribe](#) method by adding a particular observer to a subscriber list that is represented by some collection object, and it implements the [IDisposable.Dispose](#) method by removing a particular observer from the subscriber list. An observer can call these methods at any time. In addition, because the provider/observer contract does not specify who is responsible for unsubscribing after the [IObserver<T>.OnCompleted](#) callback method, the provider and observer may both try to remove the same member from the list. Because of this possibility, both the [Subscribe](#) and [Dispose](#) methods should be thread-safe. Typically, this involves using a [concurrent collection](#) or a lock. Implementations that are not thread-safe should explicitly document that they are not.

Any additional guarantees have to be specified in a layer on top of the provider/observer contract. Implementers should clearly call out when they impose additional requirements to avoid user confusion about the observer contract.

Handling Exceptions

Because of the loose coupling between a data provider and an observer, exceptions in the observer design pattern are intended to be informational. This affects how providers and observers handle exceptions in the observer design pattern.

The Provider — Calling the OnError Method

The [OnError](#) method is intended as an informational message to observers, much like the [IObserver<T>.OnNext](#) method. However, the [OnNext](#) method is designed to provide an observer with current or updated data, whereas the [OnError](#) method is designed to indicate that the provider is unable to provide valid data.

The provider should follow these best practices when handling exceptions and calling the [OnError](#) method:

- The provider must handle its own exceptions if it has any specific requirements.
- The provider should not expect or require that observers handle exceptions in any particular way.
- The provider should call the [OnError](#) method when it handles an exception that compromises its ability to provide updates. Information on such exceptions can be passed to the observer. In other cases, there is no need to notify observers of an exception.

Once the provider calls the [OnError](#) or [IObserver<T>.OnCompleted](#) method, there should be no further notifications, and the provider can unsubscribe its observers. However, the observers can also unsubscribe themselves at any time, including both before and after they receive an [OnError](#) or [IObserver<T>.OnCompleted](#) notification. The observer design pattern does not dictate whether the provider or the observer is responsible for unsubscribing; therefore, there is a possibility that both may attempt to unsubscribe. Typically, when observers unsubscribe, they are removed from a subscribers collection. In a single-threaded application, the [IDisposable.Dispose](#) implementation should ensure that an object reference is valid and that the object is a member of the subscribers collection before attempting to remove it. In a multithreaded application, a thread-safe collection object, such as a [System.Collections.Concurrent.BlockingCollection<T>](#) object, should be used.

The Observer — Implementing the OnError Method

When an observer receives an error notification from a provider, the observer should treat the exception as informational and should not be required to take any particular action.

The observer should follow these best practices when responding to an [OnError](#) method call from a provider:

- The observer should not throw exceptions from its interface implementations, such as [OnNext](#) or [OnError](#). However, if the observer does throw exceptions, it should expect these exceptions to go unhandled.
- To preserve the call stack, an observer that wishes to throw an [Exception](#) object that was passed to its [OnError](#) method should wrap the exception before throwing it. A standard exception object should be used for this purpose.

Additional Best Practices

Attempting to unregister in the `IObservable<T>.Subscribe` method may result in a null reference. Therefore, we recommend that you avoid this practice.

Although it is possible to attach an observer to multiple providers, the recommended pattern is to attach an `IObserver<T>` instance to only one `IObservable<T>` instance.

See also

- [Observer Design Pattern](#)
- [How to: Implement an Observer](#)
- [How to: Implement a Provider](#)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

How to: Implement a Provider

Article • 09/15/2021

The observer design pattern requires a division between a provider, which monitors data and sends notifications, and one or more observers, which receive notifications (callbacks) from the provider. This topic discusses how to create a provider. A related topic, [How to: Implement an Observer](#), discusses how to create an observer.

To create a provider

1. Define the data that the provider is responsible for sending to observers. Although the provider and the data that it sends to observers can be a single type, they are generally represented by different types. For example, in a temperature monitoring application, the `Temperature` structure defines the data that the provider (which is represented by the `TemperatureMonitor` class defined in the next step) monitors and to which observers subscribe.

C#

```
using System;

public struct Temperature
{
    private decimal temp;
    private DateTime tempDate;

    public Temperature(decimal temperature, DateTime dateAndTime)
    {
        this.temp = temperature;
        this.tempDate = dateAndTime;
    }

    public decimal Degrees
    { get { return this.temp; } }

    public DateTime Date
    { get { return this.tempDate; } }
}
```

2. Define the data provider, which is a type that implements the `System.IObservable<T>` interface. The provider's generic type argument is the type that the provider sends to observers. The following example defines a `TemperatureMonitor` class, which is a constructed `System.IObservable<T>` implementation with a generic type argument of `Temperature`.

C#

```
using System;
using System.Collections.Generic;

public class TemperatureMonitor : IObservable<Temperature>
{
```

3. Determine how the provider will store references to observers so that each observer can be notified when appropriate. Most commonly, a collection object such as a generic `List<T>` object is used for this purpose. The following example defines a private `List<T>` object that is instantiated in the `TemperatureMonitor` class constructor.

C#

```
using System;
using System.Collections.Generic;

public class TemperatureMonitor : IObservable<Temperature>
{
    List<IObserver<Temperature>> observers;

    public TemperatureMonitor()
    {
        observers = new List<IObserver<Temperature>>();
    }
```

4. Define an `IDisposable` implementation that the provider can return to subscribers so that they can stop receiving notifications at any time. The following example defines a nested `Unsubscriber` class that is passed a reference to the subscribers collection and to the subscriber when the class is instantiated. This code enables the subscriber to call the object's `IDisposable.Dispose` implementation to remove itself from the subscribers collection.

C#

```
private class Unsubscriber : IDisposable
{
    private List<IObserver<Temperature>> _observers;
    private IObserver<Temperature> _observer;

    public Unsubscriber(List<IObserver<Temperature>> observers,
IObserver<Temperature> observer)
    {
        this._observers = observers;
        this._observer = observer;
    }
```

```
public void Dispose()
{
    if (! (_observer == null)) _observers.Remove(_observer);
}
}
```

5. Implement the `IObservable<T>.Subscribe` method. The method is passed a reference to the `System.IObserver<T>` interface and should be stored in the object designed for that purpose in step 3. The method should then return the `IDisposable` implementation developed in step 4. The following example shows the implementation of the `Subscribe` method in the `TemperatureMonitor` class.

C#

```
public IDisposable Subscribe(IObserver<Temperature> observer)
{
    if (! observers.Contains(observer))
        observers.Add(observer);

    return new Unsubscriber(observers, observer);
}
```

6. Notify observers as appropriate by calling their `IObserver<T>.OnNext`, `IObserver<T>.OnError`, and `IObserver<T>.OnCompleted` implementations. In some cases, a provider may not call the `OnError` method when an error occurs. For example, the following `GetTemperature` method simulates a monitor that reads temperature data every five seconds and notifies observers if the temperature has changed by at least .1 degree since the previous reading. If the device does not report a temperature (that is, if its value is null), the provider notifies observers that the transmission is complete. Note that, in addition to calling each observer's `OnCompleted` method, the `GetTemperature` method clears the `List<T>` collection. In this case, the provider makes no calls to the `OnError` method of its observers.

C#

```
public void GetTemperature()
{
    // Create an array of sample data to mimic a temperature device.
    Nullable<Decimal>[] temps = {14.6m, 14.65m, 14.7m, 14.9m, 14.9m,
    15.2m, 15.25m, 15.2m,
                           15.4m, 15.45m, null };

    // Store the previous temperature, so notification is only sent
    // after at least .1 change.
    Nullable<Decimal> previous = null;
    bool start = true;
```

```

        foreach (var temp in temps) {
            System.Threading.Thread.Sleep(2500);
            if (temp.HasValue) {
                if (start || (Math.Abs(temp.Value - previous.Value) >= 0.1m ))
                {
                    Temperature tempData = new Temperature(temp.Value,
                    DateTime.Now);
                    foreach (var observer in observers)
                        observer.OnNext(tempData);
                    previous = temp;
                    if (start) start = false;
                }
            }
            else {
                foreach (var observer in observers.ToArray())
                    if (observer != null) observer.OnCompleted();

                observers.Clear();
                break;
            }
        }
    }
}

```

Example

The following example contains the complete source code for defining an `IObservable<T>` implementation for a temperature monitoring application. It includes the `Temperature` structure, which is the data sent to observers, and the `TemperatureMonitor` class, which is the `IObservable<T>` implementation.

C#

```

using System.Threading;
using System;
using System.Collections.Generic;

public class TemperatureMonitor : IObserverable<Temperature>
{
    List<IObserver<Temperature>> observers;

    public TemperatureMonitor()
    {
        observers = new List<IObserver<Temperature>>();
    }

    private class Unsubscriber : IDisposable
    {
        private List<IObserver<Temperature>> _observers;
        private IObserver<Temperature> _observer;
    }
}

```

```

    public Unsubscriber(List<IObserver<Temperature>> observers,
IObserver<Temperature> observer)
{
    this._observers = observers;
    this._observer = observer;
}

public void Dispose()
{
    if (! (_observer == null)) _observers.Remove(_observer);
}
}

public IDisposable Subscribe(IObserver<Temperature> observer)
{
    if (! observers.Contains(observer))
        observers.Add(observer);

    return new Unsubscriber(observers, observer);
}

public void GetTemperature()
{
    // Create an array of sample data to mimic a temperature device.
    Nullable<Decimal>[] temps = {14.6m, 14.65m, 14.7m, 14.9m, 14.9m,
15.2m, 15.25m, 15.2m,
                                         15.4m, 15.45m, null };

    // Store the previous temperature, so notification is only sent after
at least .1 change.
    Nullable<Decimal> previous = null;
    bool start = true;

    foreach (var temp in temps) {
        System.Threading.Thread.Sleep(2500);
        if (temp.HasValue) {
            if (start || (Math.Abs(temp.Value - previous.Value) >= 0.1m )) {
                Temperature tempData = new Temperature(temp.Value,
DateTime.Now);
                foreach (var observer in observers)
                    observer.OnNext(tempData);
                previous = temp;
                if (start) start = false;
            }
        }
        else {
            foreach (var observer in observers.ToArray())
                if (observer != null) observer.OnCompleted();

            observers.Clear();
            break;
        }
    }
}
}

```

See also

- [IObservable<T>](#)
- [Observer Design Pattern](#)
- [How to: Implement an Observer](#)
- [Observer Design Pattern Best Practices](#)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

How to: Implement an Observer

Article • 11/10/2021

The observer design pattern requires a division between an observer, which registers for notifications, and a provider, which monitors data and sends notifications to one or more observers. This topic discusses how to create an observer. A related topic, [How to: Implement a Provider](#), discusses how to create a provider.

To create an observer

1. Define the observer, which is a type that implements the `System.IObserver<T>` interface. For example, the following code defines a type named `TemperatureReporter` that is a constructed `System.IObserver<T>` implementation with a generic type argument of `Temperature`.

C#

```
public class TemperatureReporter : IObserver<Temperature>
```

2. If the observer can stop receiving notifications before the provider calls its `IObserver<T>.OnCompleted` implementation, define a private variable that will hold the `IDisposable` implementation returned by the provider's `IObservable<T>.Subscribe` method. You should also define a subscription method that calls the provider's `Subscribe` method and stores the returned `IDisposable` object. For example, the following code defines a private variable named `unsubscriber` and defines a `Subscribe` method that calls the provider's `Subscribe` method and assigns the returned object to the `unsubscriber` variable.

C#

```
public class TemperatureReporter : IObserver<Temperature>
{
    private IDisposable unsubscriber;
    private bool first = true;
    private Temperature last;

    public virtual void Subscribe(IObservable<Temperature> provider)
    {
        unsubscriber = provider.Subscribe(this);
    }
}
```

3. Define a method that enables the observer to stop receiving notifications before the provider calls its `IObserver<T>.OnCompleted` implementation, if this feature is required. The following example defines an `Unsubscribe` method.

C#

```
public virtual void Unsubscribe()
{
    subscriber.Dispose();
}
```

4. Provide implementations of the three methods defined by the `IObserver<T>` interface: `IObserver<T>.OnNext`, `IObserver<T>.OnError`, and `IObserver<T>.OnCompleted`. Depending on the provider and the needs of the application, the `OnError` and `OnCompleted` methods can be stub implementations. Note that the `OnError` method should not handle the passed `Exception` object as an exception, and the `OnCompleted` method is free to call the provider's `IDisposable.Dispose` implementation. The following example shows the `IObserver<T>` implementation of the `TemperatureReporter` class.

C#

```
public virtual void OnCompleted()
{
    Console.WriteLine("Additional temperature data will not be
transmitted.");
}

public virtual void OnError(Exception error)
{
    // Do nothing.
}

public virtual void OnNext(Temperature value)
{
    Console.WriteLine("The temperature is {0}°C at {1:g}",
value.Degrees, value.Date);
    if (first)
    {
        last = value;
        first = false;
    }
    else
    {
        Console.WriteLine("    Change: {0}° in {1:g}", value.Degrees -
last.Degrees,
value.Date.ToUniversalTime() - last.Date.ToUniversalTime());
    }
}
```

```
    }  
}
```

Example

The following example contains the complete source code for the `TemperatureReporter` class, which provides the `IObserver<T>` implementation for a temperature monitoring application.

C#

```
public class TemperatureReporter : IObserver<Temperature>  
{  
    private IDisposable unsubscriber;  
    private bool first = true;  
    private Temperature last;  
  
    public virtual void Subscribe(IObservable<Temperature> provider)  
    {  
        unsubscriber = provider.Subscribe(this);  
    }  
  
    public virtual void Unsubscribe()  
    {  
        unsubscriber.Dispose();  
    }  
  
    public virtual void OnCompleted()  
    {  
        Console.WriteLine("Additional temperature data will not be  
transmitted.");  
    }  
  
    public virtual void OnError(Exception error)  
    {  
        // Do nothing.  
    }  
  
    public virtual void OnNext(Temperature value)  
    {  
        Console.WriteLine("The temperature is {0}°C at {1:g}", value.Degrees,  
value.Date);  
        if (first)  
        {  
            last = value;  
            first = false;  
        }  
        else  
        {  
            Console.WriteLine("    Change: {0}° in {1:g}", value.Degrees -  
last.Degrees,
```

```
value.Date.ToUniversalTime() - last.Date.ToUniversalTime());
    }
}
}
```

See also

- [IObserver<T>](#)
- [Observer Design Pattern](#)
- [How to: Implement a Provider](#)
- [Observer Design Pattern Best Practices](#)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

Handling and throwing exceptions in .NET

Article • 09/15/2021

Applications must be able to handle errors that occur during execution in a consistent manner. .NET provides a model for notifying applications of errors in a uniform way: .NET operations indicate failure by throwing exceptions.

Exceptions

An exception is any error condition or unexpected behavior that is encountered by an executing program. Exceptions can be thrown because of a fault in your code or in code that you call (such as a shared library), unavailable operating system resources, unexpected conditions that the runtime encounters (such as code that can't be verified), and so on. Your application can recover from some of these conditions, but not from others. Although you can recover from most application exceptions, you can't recover from most runtime exceptions.

In .NET, an exception is an object that inherits from the [System.Exception](#) class. An exception is thrown from an area of code where a problem has occurred. The exception is passed up the stack until the application handles it or the program terminates.

Exceptions vs. traditional error-handling methods

Traditionally, a language's error-handling model relied on either the language's unique way of detecting errors and locating handlers for them, or on the error-handling mechanism provided by the operating system. The way .NET implements exception handling provides the following advantages:

- Exception throwing and handling works the same for .NET programming languages.
- Doesn't require any particular language syntax for handling exceptions, but allows each language to define its own syntax.
- Exceptions can be thrown across process and even machine boundaries.
- Exception-handling code can be added to an application to increase program reliability.

Exceptions offer advantages over other methods of error notification, such as return codes. Failures don't go unnoticed because if an exception is thrown and you don't handle it, the runtime terminates your application. Invalid values don't continue to propagate through the system as a result of code that fails to check for a failure return code.

Common exceptions

The following table lists some common exceptions with examples of what can cause them.

| Exception type | Description | Example |
|-----------------------------|--|---|
| Exception | Base class for all exceptions. | None (use a derived class of this exception). |
| IndexOutOfRangeException | Thrown by the runtime only when an array is indexed improperly. | Indexing an array outside its valid range: <code>arr[arr.Length+1]</code> |
| NullReferenceException | Thrown by the runtime only when a null object is referenced. | <code>object o = null;</code> <code>o.ToString();</code> |
| InvalidOperationException | Thrown by methods when in an invalid state. | Calling <code>Enumerator.MoveNext()</code> after removing an item from the underlying collection. |
| ArgumentException | Base class for all argument exceptions. | None (use a derived class of this exception). |
| ArgumentNullException | Thrown by methods that do not allow an argument to be null. | <code>String s = null;</code> <code>"Calculate".IndexOf(s);</code> |
| ArgumentOutOfRangeException | Thrown by methods that verify that arguments are in a given range. | <code>String s = "string";</code> <code>s.Substring(s.Length+1);</code> |

See also

- [Exception Class and Properties](#)
- [How to: Use the Try-Catch Block to Catch Exceptions](#)
- [How to: Use Specific Exceptions in a Catch Block](#)
- [How to: Explicitly Throw Exceptions](#)

- [How to: Create User-Defined Exceptions](#)
- [Using User-Filtered Exception Handlers](#)
- [How to: Use Finally Blocks](#)
- [Handling COM Interop Exceptions](#)
- [Best Practices for Exceptions](#)
- [What Every Dev needs to Know About Exceptions in the Runtime ↗](#)

 [Collaborate with us on GitHub](#)

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

Exception class and properties

Article • 09/15/2021

The [Exception](#) class is the base class from which exceptions inherit. For example, the [InvalidOperationException](#) class hierarchy is as follows:

```
Object
  |
  Exception
    |
    SystemException
      |
      InvalidOperationException
```

The [Exception](#) class has the following properties that help make understanding an exception easier.

| Property | Description |
|----------------|---|
| Name | |
| Data | An IDictionary that holds arbitrary data in key-value pairs. |
| HelpLink | Can hold a URL (or URN) to a help file that provides extensive information about the cause of an exception. |
| InnerException | This property can be used to create and preserve a series of exceptions during exception handling. You can use it to create a new exception that contains previously caught exceptions. The original exception can be captured by the second exception in the InnerException property, allowing code that handles the second exception to examine the additional information. For example, suppose you have a method that receives an argument that's improperly formatted. The code tries to read the argument, but an exception is thrown. The method catches the exception and throws a FormatException . To improve the caller's ability to determine the reason an exception is thrown, it is sometimes desirable for a method to catch an exception thrown by a helper routine and then throw an exception more indicative of the error that has occurred. A new and more meaningful exception can be created, where the inner exception reference can be set to the original exception. This more meaningful exception can then be thrown to the caller. Note that with this functionality, you can create a series of linked exceptions that ends with the exception that was thrown first. |
| Message | Provides details about the cause of an exception. |
| Source | Gets or sets the name of the application or the object that causes the error. |
| StackTrace | Contains a stack trace that can be used to determine where an error occurred. The stack trace includes the source file name and program line number if debugging information is available. |

Most of the classes that inherit from [Exception](#) do not implement additional members or provide additional functionality; they simply inherit from [Exception](#). Therefore, the most important information for an exception can be found in the hierarchy of exception classes, the exception name, and the information contained in the exception.

We recommend that you throw and catch only objects that derive from [Exception](#), but you can throw any object that derives from the [Object](#) class as an exception. Note that not all languages support throwing and catching objects that do not derive from [Exception](#).

See also

- [Exceptions](#)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

How to use the try/catch block to catch exceptions

Article • 09/15/2021

Place any code statements that might raise or throw an exception in a `try` block, and place statements used to handle the exception or exceptions in one or more `catch` blocks below the `try` block. Each `catch` block includes the exception type and can contain additional statements needed to handle that exception type.

In the following example, a [StreamReader](#) opens a file called *data.txt* and retrieves a line from the file. Since the code might throw any of three exceptions, it's placed in a `try` block. Three `catch` blocks catch the exceptions and handle them by displaying the results to the console.

C#

```
using System;
using System.IO;

public class ProcessFile
{
    public static void Main()
    {
        try
        {
            using (StreamReader sr = File.OpenText("data.txt"))
            {
                Console.WriteLine($"The first line of this file is
{sr.ReadLine()}");
            }
        }
        catch (FileNotFoundException e)
        {
            Console.WriteLine($"The file was not found: '{e}'");
        }
        catch (DirectoryNotFoundException e)
        {
            Console.WriteLine($"The directory was not found: '{e}'");
        }
        catch (IOException e)
        {
            Console.WriteLine($"The file could not be opened: '{e}'");
        }
    }
}
```

The Common Language Runtime (CLR) catches exceptions not handled by `catch` blocks.

If an exception is caught by the CLR, one of the following results may occur depending on your CLR configuration:

- A **Debug** dialog box appears.
- The program stops execution and a dialog box with exception information appears.
- An error prints out to the [standard error output stream](#).

Note

Most code can throw an exception, and some exceptions, like `OutOfMemoryException`, can be thrown by the CLR itself at any time. While applications aren't required to deal with these exceptions, be aware of the possibility when writing libraries to be used by others. For suggestions on when to set code in a `try` block, see [Best Practices for Exceptions](#).

See also

- [Exceptions](#)
- [Handling I/O errors in .NET](#)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

How to use specific exceptions in a catch block

Article • 09/15/2021

In general, it's good programming practice to catch a specific type of exception rather than use a basic `catch` statement.

When an exception occurs, it is passed up the stack and each catch block is given the opportunity to handle it. The order of catch statements is important. Put catch blocks targeted to specific exceptions before a general exception catch block or the compiler might issue an error. The proper catch block is determined by matching the type of the exception to the name of the exception specified in the catch block. If there is no specific catch block, the exception is caught by a general catch block, if one exists.

The following code example uses a `try/catch` block to catch an `InvalidCastException`. The sample creates a class called `Employee` with a single property, employee level (`Emlevel`). A method, `PromoteEmployee`, takes an object and increments the employee level. An `InvalidCastException` occurs when a `DateTime` instance is passed to the `PromoteEmployee` method.

C#

```
using System;

public class Employee
{
    //Create employee level property.
    public int Emlevel
    {
        get
        {
            return(emlevel);
        }
        set
        {
            emlevel = value;
        }
    }

    private int emlevel = 0;
}

public class Ex13
{
    public static void PromoteEmployee(Object emp)
    {
```

```
// Cast object to Employee.  
var e = (Employee) emp;  
// Increment employee level.  
e.Emlevel = e.Emlevel + 1;  
  
}  
  
static void Main()  
{  
    try  
    {  
        Object o = new Employee();  
        DateTime newYears = new DateTime(2001, 1, 1);  
        // Promote the new employee.  
        PromoteEmployee(o);  
        // Promote DateTime; results in InvalidCastException as newYears  
is not an employee instance.  
        PromoteEmployee(newYears);  
    }  
    catch (InvalidCastException e)  
    {  
        Console.WriteLine("Error passing data to PromoteEmployee method.  
" + e.Message);  
    }  
}
```

See also

- [Exceptions](#)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

Open a documentation issue

Provide product feedback

How to explicitly throw exceptions

Article • 04/22/2023

You can explicitly throw an exception using the C# [throw](#) or the Visual Basic [Throw](#) statement. You can also throw a caught exception again using the [throw](#) statement. It's good coding practice to add information to an exception that's rethrown to provide more information when debugging.

The following code example uses a [try/catch](#) block to catch a possible [FileNotFoundException](#). Following the [try](#) block is a [catch](#) block that catches the [FileNotFoundException](#) and writes a message to the console if the data file is not found. The next statement is the [throw](#) statement that throws a new [FileNotFoundException](#) and adds text information to the exception.

```
C#  
  
var fs = default(FileStream);  
try  
{  
    // Opens a text file.  
    fs = new FileStream(@"C:\temp\data.txt", FileMode.Open);  
    var sr = new StreamReader(fs);  
  
    // A value is read from the file and output to the console.  
    string? line = sr.ReadLine();  
    Console.WriteLine(line);  
}  
catch (FileNotFoundException e)  
{  
    Console.WriteLine($"[Data File Missing] {e}");  
    throw new FileNotFoundException($"[data.txt not in c:\temp directory]",  
        e);  
}  
finally  
{  
    if (fs != null)  
        fs.Close();  
}
```

See also

- [Exceptions](#)

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

How to create user-defined exceptions

Article • 08/12/2022

.NET provides a hierarchy of exception classes ultimately derived from the [Exception](#) base class. However, if none of the predefined exceptions meet your needs, you can create your own exception classes by deriving from the [Exception](#) class.

When creating your own exceptions, end the class name of the user-defined exception with the word "Exception", and implement the three common constructors, as shown in the following example. The example defines a new exception class named

`EmployeeListNotFoundException`.

The class is derived from the [Exception](#) base class and includes three constructors.

C#

```
using System;

public class EmployeeListNotFoundException : Exception
{
    public EmployeeListNotFoundException()
    {

    }

    public EmployeeListNotFoundException(string message)
        : base(message)
    {

    }

    public EmployeeListNotFoundException(string message, Exception inner)
        : base(message, inner)
    {

}
}
```

ⓘ Note

In situations where you're using remoting, you must ensure that the metadata for any user-defined exceptions is available at the server (callee) and to the client (the proxy object or caller). For more information, see [Best practices for exceptions](#).

See also

- [Exceptions](#)

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

How to create user-defined exceptions with localized exception messages

Article • 09/15/2021

In this article, you will learn how to create user-defined exceptions that are inherited from the base [Exception](#) class with localized exception messages using satellite assemblies.

Create custom exceptions

.NET contains many different exceptions that you can use. However, in some cases when none of them meets your needs, you can create your own custom exceptions.

Let's assume you want to create a `StudentNotFoundException` that contains a `StudentName` property. To create a custom exception, follow these steps:

1. Create a serializable class that inherits from [Exception](#). The class name should end in "Exception":

C#

```
[Serializable]
public class StudentNotFoundException : Exception { }
```

2. Add the default constructors:

C#

```
[Serializable]
public class StudentNotFoundException : Exception
{
    public StudentNotFoundException() { }

    public StudentNotFoundException(string message)
        : base(message) { }

    public StudentNotFoundException(string message, Exception inner)
        : base(message, inner) { }
}
```

3. Define any additional properties and constructors:

C#

```
[Serializable]
public class StudentNotFoundException : Exception
{
    public string StudentName { get; }

    public StudentNotFoundException() { }

    public StudentNotFoundException(string message)
        : base(message) { }

    public StudentNotFoundException(string message, Exception inner)
        : base(message, inner) { }

    public StudentNotFoundException(string message, string studentName)
        : this(message)
    {
        StudentName = studentName;
    }
}
```

Create localized exception messages

You have created a custom exception, and you can throw it anywhere with code like the following:

C#

```
throw new StudentNotFoundException("The student cannot be found.", "John");
```

The problem with the previous line is that "The student cannot be found." is just a constant string. In a localized application, you want to have different messages depending on user culture. [Satellite assemblies](#) are a good way to do that. A satellite assembly is a .dll that contains resources for a specific language. When you ask for a specific resources at run time, the CLR finds that resource depending on user culture. If no satellite assembly is found for that culture, the resources of the default culture are used.

To create the localized exception messages:

1. Create a new folder named *Resources* to hold the resource files.
2. Add a new resource file to it. To do that in Visual Studio, right-click the folder in **Solution Explorer**, and select **Add > New Item > Resources File**. Name the file *ExceptionMessages.resx*. This is the default resources file.

3. Add a name/value pair for your exception message, like the following image shows:

The screenshot shows a Windows application window titled "ExceptionMessages.resx". The interface includes a toolbar with "Strings", "Add Resource", "Remove Resource", and "Access Modifier: Public". A table below lists a single entry:

| | Name | Value |
|---|-----------------|------------------------------|
| ▶ | StudentNotFound | The student cannot be found. |
| * | | |

4. Add a new resource file for French. Name it *ExceptionMessages.fr-FR.resx*.
5. Add a name/value pair for the exception message again, but with a French value:

The screenshot shows a Windows application window titled "ExceptionMessages.fr-FR.resx". The interface includes a toolbar with "Strings", "Add Resource", "Remove Resource", and "Access Modifier: No code gen". A table below lists a single entry:

| | Name | Value |
|---|-----------------|-----------------------------|
| ▶ | StudentNotFound | L'étudiant est introuvable. |
| * | | |

6. After you build the project, the build output folder should contain the *fr-FR* folder with a *.dll* file, which is the satellite assembly.
7. You throw the exception with code like the following:

```
C#  
  
var resourceManager = new  
ResourceManager("FULLY_QUALIFIED_NAME_OF_RESOURCE_FILE",  
Assembly.GetExecutingAssembly());  
throw new  
StudentNotFoundException(resourceManager.GetString("StudentNotFound"),  
"John");
```

ⓘ Note

If the project name is `TestProject` and the resource file `ExceptionMessages.resx` resides in the `Resources` folder of the project, the fully qualified name of the resource file is `TestProject.Resources.ExceptionMessages`.

See also

- [How to create user-defined exceptions](#)
- [Create satellite assemblies](#)
- [base \(C# Reference\)](#)
- [this \(C# Reference\)](#)

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

How to use finally blocks

Article • 09/15/2021

When an exception occurs, execution stops and control is given to the appropriate exception handler. This often means that lines of code you expect to be executed are bypassed. Some resource cleanup, such as closing a file, needs to be done even if an exception is thrown. To do this, you can use a `finally` block. A `finally` block always executes, regardless of whether an exception is thrown.

The following code example uses a `try/catch` block to catch an [ArgumentOutOfRangeException](#). The `Main` method creates two arrays and attempts to copy one to the other. The action generates an [ArgumentOutOfRangeException](#) and the error is written to the console. The `finally` block executes regardless of the outcome of the copy action.

C#

```
using System;

class ArgumentOutOfRangeExceptionExample
{
    public static void Main()
    {
        int[] array1 = {0, 0};
        int[] array2 = {0, 0};

        try
        {
            Array.Copy(array1, array2, -1);
        }
        catch (ArgumentOutOfRangeException e)
        {
            Console.WriteLine("Error: {0}", e);
            throw;
        }
        finally
        {
            Console.WriteLine("This statement is always executed.");
        }
    }
}
```

See also

- [Exceptions](#)

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

Use user-filtered exception handlers

Article • 09/15/2021

User-filtered exception handlers catch and handle exceptions based on requirements you define for the exception. These handlers use the `catch` statement with the `when` keyword (`catch` and `When` in Visual Basic).

This technique is useful when a particular exception object corresponds to multiple errors. In this case, the object typically has a property that contains the specific error code associated with the error. You can use the error code property in the expression to select only the particular error you want to handle in that `catch` clause.

The following example illustrates the `catch/when` statement.

C#

```
try
{
    //Try statements.
}
catch (Exception ex) when (ex.Message.Contains("404"))
{
    //Catch statements.
}
```

The expression of the user-filtered clause is not restricted in any way. If an exception occurs during execution of the user-filtered expression, that exception is discarded and the filter expression is considered to have evaluated to false. In this case, the common language runtime continues the search for a handler for the current exception.

Combine the specific exception and the user-filtered clauses

A `catch` statement can contain both the specific exception and the user-filtered clauses. The runtime tests the specific exception first. If the specific exception succeeds, the runtime executes the user filter. The generic filter can contain a reference to the variable declared in the class filter. Note that the order of the two filter clauses cannot be reversed.

The following example shows a specific exception in the `catch` statement as well as the user-filtered clause using the `when` keyword.

C#

```
try
{
    //Try statements.
}
catch (System.Net.Http.HttpRequestException ex) when
(ex.Message.Contains("404"))
{
    //Catch statements.
}
```

See also

- [Exceptions](#)

 [Collaborate with us on GitHub](#)

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

 .NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

Handling COM Interop Exceptions

Article • 09/15/2021

Managed and unmanaged code can work together to handle exceptions. If a method throws an exception in managed code, the common language runtime can pass an HRESULT to a COM object. If a method fails in unmanaged code by returning a failure HRESULT, the runtime throws an exception that can be caught by managed code.

The runtime automatically maps the HRESULT from COM interop to more specific exceptions. For example, E_ACCESSDENIED becomes [UnauthorizedAccessException](#), E_OUTOFMEMORY becomes [OutOfMemoryException](#), and so on.

If the HRESULT is a custom result or if it is unknown to the runtime, the runtime passes a generic [COMException](#) to the client. The **ErrorCode** property of the [COMException](#) contains the HRESULT value.

Working with IErrorInfo

When an error is passed from COM to managed code, the runtime populates the exception object with error information. COM objects that support [IErrorInfo](#) and return HRESULTS provide this information to managed code exceptions. For example, the runtime maps the Description from the COM error to the exception's [Message](#) property. If the HRESULT provides no additional error information, the runtime fills many of the exception's properties with default values.

If a method fails in unmanaged code, an exception can be passed to a managed code segment. The topic [HRESULTS and Exceptions](#) contains a table showing how HRESULTS map to runtime exception objects.

See also

- [Exceptions](#)

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

more information, see [our contributor guide](#).



[Provide product feedback](#)

Best practices for exceptions

Article • 02/16/2023

A well-designed app handles exceptions and errors to prevent app crashes. This article describes best practices for handling and creating exceptions.

Use try/catch/finally blocks to recover from errors or release resources

Use `try`/`catch` blocks around code that can potentially generate an exception, and your code can recover from that exception. In `catch` blocks, always order exceptions from the most derived to the least derived. All exceptions derive from the [Exception](#) class. More derived exceptions aren't handled by a catch clause that's preceded by a catch clause for a base exception class. When your code can't recover from an exception, don't catch that exception. Enable methods further up the call stack to recover if possible.

Clean up resources that are allocated with either `using` statements or `finally` blocks. Prefer `using` statements to automatically clean up resources when exceptions are thrown. Use `finally` blocks to clean up resources that don't implement [IDisposable](#). Code in a `finally` clause is almost always executed even when exceptions are thrown.

Handle common conditions without throwing exceptions

For conditions that are likely to occur but might trigger an exception, consider handling them in a way that will avoid the exception. For example, if you try to close a connection that's already closed, you'll get an `InvalidOperationException`. You can avoid that by using an `if` statement to check the connection state before trying to close it.

C#

```
if (conn.State != ConnectionState.Closed)
{
    conn.Close();
}
```

If you don't check the connection state before closing, you can catch the `InvalidOperationException` exception.

C#

```
try
{
    conn.Close();
}
catch (InvalidOperationException ex)
{
    Console.WriteLine(ex.GetType().FullName);
    Console.WriteLine(ex.Message);
}
```

The method to choose depends on how often you expect the event to occur.

- Use exception handling if the event doesn't occur often, that is, if the event is truly exceptional and indicates an error, such as an unexpected end-of-file. When you use exception handling, less code is executed in normal conditions.
- Check for error conditions in code if the event happens routinely and could be considered part of normal execution. When you check for common error conditions, less code is executed because you avoid exceptions.

Design classes so that exceptions can be avoided

A class can provide methods or properties that enable you to avoid making a call that would trigger an exception. For example, a [FileStream](#) class provides methods that help determine whether the end of the file has been reached. These methods can be used to avoid the exception that's thrown if you read past the end of the file. The following example shows how to read to the end of a file without triggering an exception:

C#

```
class FileRead
{
    public void ReadAll(FileStream fileToRead)
    {
        // This if statement is optional
        // as it is very unlikely that
        // the stream would ever be null.
        if (fileToRead == null)
        {
            throw new ArgumentNullException();
        }

        int b;
```

```
// Set the stream position to the beginning of the file.  
fileToRead.Seek(0, SeekOrigin.Begin);  
  
// Read each byte to the end of the file.  
for (int i = 0; i < fileToRead.Length; i++)  
{  
    b = fileToRead.ReadByte();  
    Console.WriteLine(b.ToString());  
    // Or do something else with the byte.  
}  
}  
}
```

Another way to avoid exceptions is to return null (or default) for most common error cases instead of throwing an exception. A common error case can be considered a normal flow of control. By returning null (or default) in these cases, you minimize the performance impact to an app.

For value types, whether to use `Nullable<T>` or default as your error indicator is something to consider for your app. By using `Nullable<Guid>`, `default` becomes `null` instead of `Guid.Empty`. Sometimes, adding `Nullable<T>` can make it clearer when a value is present or absent. Other times, adding `Nullable<T>` can create extra cases to check that aren't necessary and only serve to create potential sources of errors.

Throw exceptions instead of returning an error code

Exceptions ensure that failures don't go unnoticed because the calling code didn't check a return code.

Use the predefined .NET exception types

Introduce a new exception class only when a predefined one doesn't apply. For example:

- If a property set or method call isn't appropriate given the object's current state, throw an `InvalidOperationException` exception.
- If invalid parameters are passed, throw an `ArgumentException` exception or one of the predefined classes that derive from `ArgumentException`.

End exception class names with the word Exception

When a custom exception is necessary, name it appropriately and derive it from the [Exception](#) class. For example:

```
C#
```

```
public class MyFileNotFoundException : Exception  
{  
}
```

Include three constructors in custom exception classes

Use at least the three common constructors when creating your own exception classes: the parameterless constructor, a constructor that takes a string message, and a constructor that takes a string message and an inner exception.

- [Exception\(\)](#), which uses default values.
- [Exception\(String\)](#), which accepts a string message.
- [Exception\(String, Exception\)](#), which accepts a string message and an inner exception.

For an example, see [How to: Create User-Defined Exceptions](#).

Ensure that exception data is available when code executes remotely

When you create user-defined exceptions, ensure that the metadata for the exceptions is available to code that's executing remotely.

For example, on .NET implementations that support app domains, exceptions might occur across app domains. Suppose app domain A creates app domain B, which executes code that throws an exception. For app domain A to properly catch and handle the exception, it must be able to find the assembly that contains the exception thrown by app domain B. If app domain B throws an exception that is contained in an assembly under its application base, but not under app domain A's application base, app domain A won't be able to find the exception, and the common language runtime will throw a [FileNotFoundException](#) exception. To avoid this situation, you can deploy the assembly that contains the exception information in either of two ways:

- Put the assembly into a common application base shared by both app domains.

- If the domains don't share a common application base, sign the assembly that contains the exception information with a strong name and deploy the assembly into the global assembly cache.

Use grammatically correct error messages

Write clear sentences and include ending punctuation. Each sentence in the string assigned to the [Exception.Message](#) property should end in a period. For example, "The log table has overflowed." would be an appropriate message string.

Include a localized string message in every exception

The error message the user sees is derived from the [Exception.Message](#) property of the exception that was thrown, and not from the name of the exception class. Typically, you assign a value to the [Exception.Message](#) property by passing the message string to the `message` argument of an [Exception constructor](#).

For localized applications, you should provide a localized message string for every exception that your application can throw. You use resource files to provide localized error messages. For information on localizing applications and retrieving localized strings, see the following articles:

- [How to: Create user-defined exceptions with localized exception messages](#)
- [Resources in .NET apps](#)
- [System.Resources.ResourceManager](#)

In custom exceptions, provide additional properties as needed

Provide additional properties for an exception (in addition to the custom message string) only when there's a programmatic scenario where the additional information is useful. For example, the [FileNotFoundException](#) provides the [FileName](#) property.

Place throw statements so that the stack trace will be helpful

The stack trace begins at the statement where the exception is thrown and ends at the `catch` statement that catches the exception.

Use exception builder methods

It's common for a class to throw the same exception from different places in its implementation. To avoid excessive code, use helper methods that create the exception and return it. For example:

C#

```
class FileReader
{
    private string fileName;

    public FileReader(string path)
    {
        fileName = path;
    }

    public byte[] Read(int bytes)
    {
        byte[] results = FileUtils.ReadFromFile(fileName, bytes);
        if (results == null)
        {
            throw NewFileIOException();
        }
        return results;
    }

    FileReaderException NewFileIOException()
    {
        string description = "My NewFileIOException Description";

        return new FileReaderException(description);
    }
}
```

In some cases, it's more appropriate to use the exception's constructor to build the exception. An example is a global exception class such as [ArgumentException](#).

Restore state when methods don't complete due to exceptions

Callers should be able to assume that there are no side effects when an exception is thrown from a method. For example, if you have code that transfers money by

withdrawing from one account and depositing in another account, and an exception is thrown while executing the deposit, you don't want the withdrawal to remain in effect.

C#

```
public void TransferFunds(Account from, Account to, decimal amount)
{
    from.Withdrawal(amount);
    // If the deposit fails, the withdrawal shouldn't remain in effect.
    to.Deposit(amount);
}
```

The preceding method doesn't directly throw any exceptions. However, you must write the method so that the withdrawal is reversed if the deposit operation fails.

One way to handle this situation is to catch any exceptions thrown by the deposit transaction and roll back the withdrawal.

C#

```
private static void TransferFunds(Account from, Account to, decimal amount)
{
    string withdrawalTrxID = from.Withdrawal(amount);
    try
    {
        to.Deposit(amount);
    }
    catch
    {
        from.RollbackTransaction(withdrawalTrxID);
        throw;
    }
}
```

This example illustrates the use of `throw` to rethrow the original exception, making it easier for callers to see the real cause of the problem without having to examine the `InnerException` property. An alternative is to throw a new exception and include the original exception as the inner exception.

C#

```
catch (Exception ex)
{
    from.RollbackTransaction(withdrawalTrxID);
    throw new TransferFundsException("Withdrawal failed.", innerException:
ex)
{
    From = from,
```

```
    To = to,
    Amount = amount
};

}
```

Capture exceptions to rethrow later

To capture an exception and preserve its callstack to be able to rethrow it later, use the [System.Runtime.ExceptionServices.ExceptionDispatchInfo](#) class. This class provides the following methods and properties (among others):

- Use [ExceptionDispatchInfo.Capture\(Exception\)](#) to capture an exception and call stack.
- Use [ExceptionDispatchInfo.Throw\(\)](#) to restore the state that was saved when the exception was captured and rethrow the captured exception.
- Use the [ExceptionDispatchInfo.SourceException](#) property to inspect the captured exception.

The following example shows how the [ExceptionDispatchInfo](#) class can be used, and what the output might look like.

C#

```
ExceptionDispatchInfo? edi = null;
try
{
    var txt = File.ReadAllText(@"C:\temp\file.txt");
}
catch (FileNotFoundException e)
{
    edi = ExceptionDispatchInfo.Capture(e);
}

// ...

Console.WriteLine("I was here.");

if (edi is not null)
    edi.Throw();
```

If the file in the example code doesn't exist, the following output is produced:

Output

```
I was here.
Unhandled exception. System.IO.FileNotFoundException: Could not find file
'C:\temp\file.txt'.
```

```
File name: 'C:\temp\file.txt'
   at Microsoft.Win32.SafeHandles.SafeFileHandle.CreateFile(String fullPath,
 FileMode mode, FileAccess access, FileShare share, FileOptions options)
   at Microsoft.Win32.SafeHandles.SafeFileHandle.Open(String fullPath,
 FileMode mode, FileAccess access, FileShare share, FileOptions options,
 Int64 preallocationSize, Nullable`1 unixCreateMode)
   at System.IO.Strategies.OSFileStreamStrategy..ctor(String path, FileMode
 mode, FileAccess access, FileShare share, FileOptions options, Int64
 preallocationSize, Nullable`1 unixCreateMode)
   at System.IO.Strategies.FileStreamHelpers.ChooseStrategyCore(String path,
 FileMode mode, FileAccess access, FileShare share, FileOptions options,
 Int64 preallocationSize, Nullable`1 unixCreateMode)
   at System.IO.StreamReader.ValidateArgsAndOpenPath(String path, Encoding
 encoding, Int32 bufferSize)
   at System.IO.File.ReadAllText(String path, Encoding encoding)
   at Example.ProcessFile.Main() in C:\repos\ConsoleApp1\Program.cs:line 12
--- End of stack trace from previous location ---
   at Example.ProcessFile.Main() in C:\repos\ConsoleApp1\Program.cs:line 24
```

See also

- [Exceptions](#)

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

[.NET feedback](#)

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

System.AccessViolationException class

Article • 01/04/2024

This article provides supplementary remarks to the reference documentation for this API.

An access violation occurs in unmanaged or unsafe code when the code attempts to read or write to memory that has not been allocated, or to which it does not have access. This usually occurs because a pointer has a bad value. Not all reads or writes through bad pointers lead to access violations, so an access violation usually indicates that several reads or writes have occurred through bad pointers, and that memory might be corrupted. Thus, access violations almost always indicate serious programming errors. An [AccessViolationException](#) clearly identifies these serious errors.

In programs consisting entirely of verifiable managed code, all references are either valid or null, and access violations are impossible. Any operation that attempts to reference a null reference in verifiable code throws a [NullReferenceException](#) exception. An [AccessViolationException](#) occurs only when verifiable managed code interacts with unmanaged code or with unsafe managed code.

Troubleshoot AccessViolationException exceptions

An [AccessViolationException](#) exception can occur only in unsafe managed code or when verifiable managed code interacts with unmanaged code:

- An access violation that occurs in unsafe managed code can be expressed as either a [NullReferenceException](#) exception or an [AccessViolationException](#) exception, depending on the platform.
- An access violation in unmanaged code that bubbles up to managed code is always wrapped in an [AccessViolationException](#) exception.

In either case, you can identify and correct the cause of the [AccessViolationException](#) exception as follows:

- Make sure that the memory that you're attempting to access has been allocated. An [AccessViolationException](#) exception is always thrown by an attempt to access protected memory—that is, to access memory that's not allocated or that's not owned by a process.

Automatic memory management is one of the services that the .NET runtime provides. If managed code provides the same functionality as your unmanaged code, you may wish to move to managed code to take advantage of this functionality. For more information, see [Automatic Memory Management](#).

- Make sure that the memory that you're attempting to access has not been corrupted. If several read or write operations have occurred through bad pointers, memory might be corrupted. This typically occurs when reading or writing to addresses outside of a predefined buffer.

AccessViolationException and try/catch blocks

[AccessViolationException](#) exceptions thrown by the .NET runtime aren't handled by the `catch` statement in a structured exception handler if the exception occurs outside of the memory reserved by the runtime. To handle such an [AccessViolationException](#) exception, apply the [HandleProcessCorruptedStateExceptionsAttribute](#) attribute to the method in which the exception is thrown. This change does not affect [AccessViolationException](#) exceptions thrown by user code, which can continue to be caught by a `catch` statement.

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

.NET is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

System.Exception class

Article • 01/04/2024

This article provides supplementary remarks to the reference documentation for this API.

The [Exception](#) class is the base class for all exceptions. When an error occurs, either the system or the currently executing application reports it by throwing an exception that contains information about the error. After an exception is thrown, it is handled by the application or by the default exception handler.

Errors and exceptions

Run-time errors can occur for a variety of reasons. However, not all errors should be handled as exceptions in your code. Here are some categories of errors that can occur at run time and the appropriate ways to respond to them.

- **Usage errors.** A usage error represents an error in program logic that can result in an exception. However, the error should be addressed not through exception handling but by modifying the faulty code. For example, the override of the [Object.Equals\(Object\)](#) method in the following example assumes that the `obj` argument must always be non-null.

C#

```
using System;

public class Person1
{
    private string _name;

    public string Name
    {
        get { return _name; }
        set { _name = value; }
    }

    public override int GetHashCode()
    {
        return this.Name.GetHashCode();
    }

    public override bool Equals(object obj)
    {
        // This implementation contains an error in program logic:
        // It assumes that the obj argument is not null.
    }
}
```

```

        Person1 p = (Person1) obj;
        return this.Name.Equals(p.Name);
    }
}

public class UsageErrorsEx1
{
    public static void Main()
    {
        Person1 p1 = new Person1();
        p1.Name = "John";
        Person1 p2 = null;

        // The following throws a NullReferenceException.
        Console.WriteLine("p1 = p2: {0}", p1.Equals(p2));
    }
}

```

The [NullReferenceException](#) exception that results when `obj` is `null` can be eliminated by modifying the source code to explicitly test for null before calling the [Object.Equals](#) override and then re-compiling. The following example contains the corrected source code that handles a `null` argument.

C#

```

using System;

public class Person2
{
    private string _name;

    public string Name
    {
        get { return _name; }
        set { _name = value; }
    }

    public override int GetHashCode()
    {
        return this.Name.GetHashCode();
    }

    public override bool Equals(object obj)
    {
        // This implementation handles a null obj argument.
        Person2 p = obj as Person2;
        if (p == null)
            return false;
        else
            return this.Name.Equals(p.Name);
    }
}

```

```

public class UsageErrorsEx2
{
    public static void Main()
    {
        Person2 p1 = new Person2();
        p1.Name = "John";
        Person2 p2 = null;

        Console.WriteLine("p1 = p2: {0}", p1.Equals(p2));
    }
}

// The example displays the following output:
//      p1 = p2: False

```

Instead of using exception handling for usage errors, you can use the [Debug.Assert](#) method to identify usage errors in debug builds, and the [Trace.Assert](#) method to identify usage errors in both debug and release builds. For more information, see [Assertions in Managed Code](#).

- **Program errors.** A program error is a run-time error that cannot necessarily be avoided by writing bug-free code.

In some cases, a program error may reflect an expected or routine error condition. In this case, you may want to avoid using exception handling to deal with the program error and instead retry the operation. For example, if the user is expected to input a date in a particular format, you can parse the date string by calling the [DateTime.TryParseExact](#) method, which returns a [Boolean](#) value that indicates whether the parse operation succeeded, instead of using the [DateTime.ParseExact](#) method, which throws a [FormatException](#) exception if the date string cannot be converted to a [DateTime](#) value. Similarly, if a user tries to open a file that does not exist, you can first call the [File.Exists](#) method to check whether the file exists and, if it does not, prompt the user whether they want to create it.

In other cases, a program error reflects an unexpected error condition that can be handled in your code. For example, even if you've checked to ensure that a file exists, it may be deleted before you can open it, or it may be corrupted. In that case, trying to open the file by instantiating a [StreamReader](#) object or calling the [Open](#) method may throw a [FileNotFoundException](#) exception. In these cases, you should use exception handling to recover from the error.

- **System failures.** A system failure is a run-time error that cannot be handled programmatically in a meaningful way. For example, any method can throw an [OutOfMemoryException](#) exception if the common language runtime is unable to allocate additional memory. Ordinarily, system failures are not handled by using

exception handling. Instead, you may be able to use an event such as [AppDomain.UnhandledException](#) and call the [Environment.FailFast](#) method to log exception information and notify the user of the failure before the application terminates.

Try/catch blocks

The common language runtime provides an exception handling model that is based on the representation of exceptions as objects, and the separation of program code and exception handling code into `try` blocks and `catch` blocks. There can be one or more `catch` blocks, each designed to handle a particular type of exception, or one block designed to catch a more specific exception than another block.

If an application handles exceptions that occur during the execution of a block of application code, the code must be placed within a `try` statement and is called a `try` block. Application code that handles exceptions thrown by a `try` block is placed within a `catch` statement and is called a `catch` block. Zero or more `catch` blocks are associated with a `try` block, and each `catch` block includes a type filter that determines the types of exceptions it handles.

When an exception occurs in a `try` block, the system searches the associated `catch` blocks in the order they appear in application code, until it locates a `catch` block that handles the exception. A `catch` block handles an exception of type `T` if the type filter of the catch block specifies `T` or any type that `T` derives from. The system stops searching after it finds the first `catch` block that handles the exception. For this reason, in application code, a `catch` block that handles a type must be specified before a `catch` block that handles its base types, as demonstrated in the example that follows this section. A catch block that handles `System.Exception` is specified last.

If none of the `catch` blocks associated with the current `try` block handle the exception, and the current `try` block is nested within other `try` blocks in the current call, the `catch` blocks associated with the next enclosing `try` block are searched. If no `catch` block for the exception is found, the system searches previous nesting levels in the current call. If no `catch` block for the exception is found in the current call, the exception is passed up the call stack, and the previous stack frame is searched for a `catch` block that handles the exception. The search of the call stack continues until the exception is handled or until no more frames exist on the call stack. If the top of the call stack is reached without finding a `catch` block that handles the exception, the default exception handler handles it and the application terminates.

F# try..with expression

F# does not use `catch` blocks. Instead, a raised exception is pattern matched using a single `with` block. As this is an expression, rather than a statement, all paths must return the same type. To learn more, see [The try...with Expression](#).

Exception type features

Exception types support the following features:

- Human-readable text that describes the error. When an exception occurs, the runtime makes a text message available to inform the user of the nature of the error and to suggest action to resolve the problem. This text message is held in the [Message](#) property of the exception object. During the creation of the exception object, you can pass a text string to the constructor to describe the details of that particular exception. If no error message argument is supplied to the constructor, the default error message is used. For more information, see the [Message](#) property.
- The state of the call stack when the exception was thrown. The [StackTrace](#) property carries a stack trace that can be used to determine where the error occurs in the code. The stack trace lists all the called methods and the line numbers in the source file where the calls are made.

Exception class properties

The [Exception](#) class includes a number of properties that help identify the code location, the type, the help file, and the reason for the exception: [StackTrace](#), [InnerException](#), [Message](#), [HelpLink](#), [HResult](#), [Source](#), [TargetSite](#), and [Data](#).

When a causal relationship exists between two or more exceptions, the [InnerException](#) property maintains this information. The outer exception is thrown in response to this inner exception. The code that handles the outer exception can use the information from the earlier inner exception to handle the error more appropriately. Supplementary information about the exception can be stored as a collection of key/value pairs in the [Data](#) property.

The error message string that is passed to the constructor during the creation of the exception object should be localized and can be supplied from a resource file by using the [ResourceManager](#) class. For more information about localized resources, see the [Creating Satellite Assemblies](#) and [Packaging and Deploying Resources](#) topics.

To provide the user with extensive information about why the exception occurred, the [HelpLink](#) property can hold a URL (or URN) to a help file.

The [Exception](#) class uses the HRESULT `COR_E_EXCEPTION`, which has the value 0x80131500.

For a list of initial property values for an instance of the [Exception](#) class, see the [Exception constructors](#).

Performance considerations

Throwing or handling an exception consumes a significant amount of system resources and execution time. Throw exceptions only to handle truly extraordinary conditions, not to handle predictable events or flow control. For example, in some cases, such as when you're developing a class library, it's reasonable to throw an exception if a method argument is invalid, because you expect your method to be called with valid parameters. An invalid method argument, if it is not the result of a usage error, means that something extraordinary has occurred. Conversely, do not throw an exception if user input is invalid, because you can expect users to occasionally enter invalid data. Instead, provide a retry mechanism so users can enter valid input. Nor should you use exceptions to handle usage errors. Instead, use [assertions](#) to identify and correct usage errors.

In addition, do not throw an exception when a return code is sufficient; do not convert a return code to an exception; and do not routinely catch an exception, ignore it, and then continue processing.

Re-throw an exception

In many cases, an exception handler simply wants to pass the exception on to the caller. This most often occurs in:

- A class library that in turn wraps calls to methods in the .NET class library or other class libraries.
- An application or library that encounters a fatal exception. The exception handler can log the exception and then re-throw the exception.

The recommended way to re-throw an exception is to simply use the [throw](#) statement in C#, the [reraise](#) function in F#, and the [Throw](#) statement in Visual Basic without including an expression. This ensures that all call stack information is preserved when the exception is propagated to the caller. The following example illustrates this. A string

extension method, `FindOccurrences`, wraps one or more calls to `String.IndexOf(String, Int32)` without validating its arguments beforehand.

C#

```
using System;
using System.Collections.Generic;

public static class Library1
{
    public static int[] FindOccurrences(this String s, String f)
    {
        var indexes = new List<int>();
        int currentIndex = 0;
        try
        {
            while (currentIndex >= 0 && currentIndex < s.Length)
            {
                currentIndex = s.IndexOf(f, currentIndex);
                if (currentIndex >= 0)
                {
                    indexes.Add(currentIndex);
                    currentIndex++;
                }
            }
        }
        catch (ArgumentNullException)
        {
            // Perform some action here, such as logging this exception.

            throw;
        }
        return indexes.ToArray();
    }
}
```

A caller then calls `FindOccurrences` twice. In the second call to `FindOccurrences`, the caller passes a `null` as the search string, which causes the `String.IndexOf(String, Int32)` method to throw an `ArgumentNullException` exception. This exception is handled by the `FindOccurrences` method and passed back to the caller. Because the `throw` statement is used with no expression, the output from the example shows that the call stack is preserved.

C#

```
public class RethrowEx1
{
    public static void Main()
    {
        String s = "It was a cold day when...";
```

```

        int[] indexes = s.FindOccurrences("a");
        ShowOccurrences(s, "a", indexes);
        Console.WriteLine();

        String toFind = null;
        try
        {
            indexes = s.FindOccurrences(toFind);
            ShowOccurrences(s, toFind, indexes);
        }
        catch (ArgumentNullException e)
        {
            Console.WriteLine("An exception ({0}) occurred.",
                e.GetType().Name);
            Console.WriteLine("Message:\n    {0}\n", e.Message);
            Console.WriteLine("Stack Trace:\n    {0}\n", e.StackTrace);
        }
    }

    private static void ShowOccurrences(String s, String toFind, int[]
indexes)
    {
        Console.Write('{0}' occurs at the following character positions: ",
            toFind);
        for (int ctr = 0; ctr < indexes.Length; ctr++)
            Console.Write("{0}{1}", indexes[ctr],
                ctr == indexes.Length - 1 ? "" : ", ");
        Console.WriteLine();
    }
}

// The example displays the following output:
//      'a' occurs at the following character positions: 4, 7, 15
//
//      An exception (ArgumentNullException) occurred.
//      Message:
//          Value cannot be null.
//      Parameter name: value
//
//      Stack Trace:
//          at System.String.IndexOf(String value, Int32 startIndex, Int32
count, Stri
//      ngComparison comparisonType)
//          at Library.FindOccurrences(String s, String f)
//          at Example.Main()

```

In contrast, if the exception is re-thrown by using this statement:

C#

```
throw e;
```

...then the full call stack is not preserved, and the example would generate the following output:

Output

```
'a' occurs at the following character positions: 4, 7, 15

An exception (ArgumentNullException) occurred.
Message:
    Value cannot be null.
Parameter name: value

Stack Trace:
    at Library.FindOccurrences(String s, String f)
    at Example.Main()
```

A slightly more cumbersome alternative is to throw a new exception, and to preserve the original exception's call stack information in an inner exception. The caller can then use the new exception's [InnerException](#) property to retrieve stack frame and other information about the original exception. In this case, the throw statement is:

C#

```
throw new ArgumentNullException("You must supply a search string.", e);
```

The user code that handles the exception has to know that the [InnerException](#) property contains information about the original exception, as the following exception handler illustrates.

C#

```
try
{
    indexes = s.FindOccurrences(toFind);
    ShowOccurrences(s, toFind, indexes);
}
catch (ArgumentNullException e)
{
    Console.WriteLine("An exception ({0}) occurred.",
                      e.GetType().Name);
    Console.WriteLine("    Message:\n{0}", e.Message);
    Console.WriteLine("    Stack Trace:\n    {0}", e.StackTrace);
    Exception ie = e.InnerException;
    if (ie != null)
    {
        Console.WriteLine("    The Inner Exception:");
        Console.WriteLine("        Exception Name: {0}", ie.GetType().Name);
        Console.WriteLine("        Message: {0}\n", ie.Message);
        Console.WriteLine("        Stack Trace:\n    {0}\n", ie.StackTrace);
    }
}
```

```
        }

    // The example displays the following output:
    //      'a' occurs at the following character positions: 4, 7, 15
    //

    //      An exception (ArgumentNullException) occurred.
    //          Message: You must supply a search string.
    //

    //      Stack Trace:
    //          at Library.FindOccurrences(String s, String f)
    //          at Example.Main()

    //

    //      The Inner Exception:
    //          Exception Name: ArgumentNullException
    //          Message: Value cannot be null.
    //          Parameter name: value

    //

    //      Stack Trace:
    //          at System.String.IndexOf(String value, Int32 startIndex, Int32
    count, Stri
    // ngComparison comparisonType)
    //      at Library.FindOccurrences(String s, String f)
```

Choose standard exceptions

When you have to throw an exception, you can often use an existing exception type in .NET instead of implementing a custom exception. You should use a standard exception type under these two conditions:

- You're throwing an exception that is caused by a usage error (that is, by an error in program logic made by the developer who is calling your method). Typically, you would throw an exception such as [ArgumentException](#), [ArgumentNullException](#), [InvalidOperationException](#), or [NotSupportedException](#). The string you supply to the exception object's constructor when instantiating the exception object should describe the error so that the developer can fix it. For more information, see the [Message](#) property.
- You're handling an error that can be communicated to the caller with an existing .NET exception. You should throw the most derived exception possible. For example, if a method requires an argument to be a valid member of an enumeration type, you should throw an [InvalidEnumArgumentException](#) (the most derived class) rather than an [ArgumentException](#).

The following table lists common exception types and the conditions under which you would throw them.

| Exception | Condition |
|-------------------------------|---|
| ArgumentException | A non-null argument that is passed to a method is invalid. |
| ArgumentNullException | An argument that is passed to a method is <code>null</code> . |
| ArgumentOutOfRangeException | An argument is outside the range of valid values. |
| DirectoryNotFoundException | Part of a directory path is not valid. |
| DivideByZeroException | The denominator in an integer or <code>Decimal</code> division operation is zero. |
| DriveNotFoundException | A drive is unavailable or does not exist. |
| FileNotFoundException | A file does not exist. |
| FormatException | A value is not in an appropriate format to be converted from a string by a conversion method such as <code>Parse</code> . |
| IndexOutOfRangeException | An index is outside the bounds of an array or collection. |
| InvalidOperationException | A method call is invalid in an object's current state. |
| KeyNotFoundException | The specified key for accessing a member in a collection cannot be found. |
| NotImplementedException | A method or operation is not implemented. |
| NotSupportedException | A method or operation is not supported. |
| ObjectDisposedException | An operation is performed on an object that has been disposed. |
| OverflowException | An arithmetic, casting, or conversion operation results in an overflow. |
| PathTooLongException | A path or file name exceeds the maximum system-defined length. |
| PlatformNotSupportedException | The operation is not supported on the current platform. |
| RankException | An array with the wrong number of dimensions is passed to a method. |
| TimeoutException | The time interval allotted to an operation has expired. |
| UriFormatException | An invalid Uniform Resource Identifier (URI) is used. |

Implement custom exceptions

In the following cases, using an existing .NET exception to handle an error condition is not adequate:

- When the exception reflects a unique program error that cannot be mapped to an existing .NET exception.
- When the exception requires handling that is different from the handling that is appropriate for an existing .NET exception, or the exception must be disambiguated from a similar exception. For example, if you throw an [ArgumentOutOfRangeException](#) exception when parsing the numeric representation of a string that is out of range of the target integral type, you would not want to use the same exception for an error that results from the caller not supplying the appropriate constrained values when calling the method.

The [Exception](#) class is the base class of all exceptions in .NET. Many derived classes rely on the inherited behavior of the members of the [Exception](#) class; they do not override the members of [Exception](#), nor do they define any unique members.

To define your own exception class:

1. Define a class that inherits from [Exception](#). If necessary, define any unique members needed by your class to provide additional information about the exception. For example, the [ArgumentException](#) class includes a [ParamName](#) property that specifies the name of the parameter whose argument caused the exception, and the [RegexMatchTimeoutException](#) property includes a [MatchTimeout](#) property that indicates the time-out interval.
2. If necessary, override any inherited members whose functionality you want to change or modify. Note that most existing derived classes of [Exception](#) do not override the behavior of inherited members.
3. Determine whether your custom exception object is serializable. Serialization enables you to save information about the exception and permits exception information to be shared by a server and a client proxy in a remoting context. To make the exception object serializable, mark it with the [SerializableAttribute](#) attribute.
4. Define the constructors of your exception class. Typically, exception classes have one or more of the following constructors:
 - [Exception\(\)](#), which uses default values to initialize the properties of a new exception object.

- `Exception(String)`, which initializes a new exception object with a specified error message.
- `Exception(String, Exception)`, which initializes a new exception object with a specified error message and inner exception.
- `Exception(SerializationInfo, StreamingContext)`, which is a `protected` constructor that initializes a new exception object from serialized data. You should implement this constructor if you've chosen to make your exception object serializable.

The following example illustrates the use of a custom exception class. It defines a `NotPrimeException` exception that is thrown when a client tries to retrieve a sequence of prime numbers by specifying a starting number that is not prime. The exception defines a new property, `NonPrime`, that returns the non-prime number that caused the exception. Besides implementing a protected parameterless constructor and a constructor with `SerializationInfo` and `StreamingContext` parameters for serialization, the `NotPrimeException` class defines three additional constructors to support the `NonPrime` property. Each constructor calls a base class constructor in addition to preserving the value of the non-prime number. The `NotPrimeException` class is also marked with the `SerializableAttribute` attribute.

C#

```
using System;
using System.Runtime.Serialization;

[Serializable()]
public class NotPrimeException : Exception
{
    private int notAPrime;

    protected NotPrimeException()
        : base()
    { }

    public NotPrimeException(int value) :
        base(String.Format("{0} is not a prime number.", value))
    {
        notAPrime = value;
    }

    public NotPrimeException(int value, string message)
        : base(message)
    {
        notAPrime = value;
    }
}
```

```

    public NotPrimeException(int value, string message, Exception
innerException) :
        base(message, innerException)
    {
        notAPrime = value;
    }

    protected NotPrimeException(SerializationInfo info,
                                StreamingContext context)
        : base(info, context)
    { }

    public int NonPrime
    { get { return notAPrime; } }
}

```

The `PrimeNumberGenerator` class shown in the following example uses the Sieve of Eratosthenes to calculate the sequence of prime numbers from 2 to a limit specified by the client in the call to its class constructor. The `GetPrimesFrom` method returns all prime numbers that are greater than or equal to a specified lower limit, but throws a `NotPrimeException` if that lower limit is not a prime number.

C#

```

using System;
using System.Collections.Generic;

[Serializable]
public class PrimeNumberGenerator
{
    private const int START = 2;
    private int maxUpperBound = 10000000;
    private int upperBound;
    private bool[] primeTable;
    private List<int> primes = new List<int>();

    public PrimeNumberGenerator(int upperBound)
    {
        if (upperBound > maxUpperBound)
        {
            string message = String.Format(
                "{0} exceeds the maximum upper bound of {1}.",
                upperBound, maxUpperBound);
            throw new ArgumentOutOfRangeException(message);
        }
        this.upperBound = upperBound;
        // Create array and mark 0, 1 as not prime (True).
        primeTable = new bool[upperBound + 1];
        primeTable[0] = true;
        primeTable[1] = true;
    }
}

```

```

// Use Sieve of Eratosthenes to determine prime numbers.
for (int ctr = START; ctr <= (int)Math.Ceiling(Math.Sqrt(upperBound));
     ctr++)
{
    if (primeTable[ctr]) continue;

    for (int multiplier = ctr; multiplier <= upperBound / ctr;
multiplier++)
        if (ctr * multiplier <= upperBound) primeTable[ctr * multiplier]
= true;
}
// Populate array with prime number information.
int index = START;
while (index != -1)
{
    index = Array.FindIndex(primeTable, index, (flag) => !flag);
    if (index >= 1)
    {
        primes.Add(index);
        index++;
    }
}
}

public int[] GetAllPrimes()
{
    return primes.ToArray();
}

public int[] GetPrimesFrom(int prime)
{
    int start = primes.FindIndex((value) => value == prime);
    if (start < 0)
        throw new NotPrimeException(prime, String.Format("{0} is not a
prime number.", prime));
    else
        return primes.FindAll((value) => value >= prime).ToArray();
}
}

```

The following example makes two calls to the `GetPrimesFrom` method with non-prime numbers, one of which crosses application domain boundaries. In both cases, the exception is thrown and successfully handled in client code.

C#

```

using System;
using System.Reflection;

class Example1
{
    public static void Main()

```

```

{
    int limit = 10000000;
    PrimeNumberGenerator primes = new PrimeNumberGenerator(limit);
    int start = 1000001;
    try
    {
        int[] values = primes.GetPrimesFrom(start);
        Console.WriteLine("There are {0} prime numbers from {1} to {2}",
                          start, limit);
    }
    catch (NotPrimeException e)
    {
        Console.WriteLine("{0} is not prime", e.NonPrime);
        Console.WriteLine(e);
        Console.WriteLine("-----");
    }
}

AppDomain domain = AppDomain.CreateDomain("Domain2");
PrimeNumberGenerator gen =
(PrimeNumberGenerator)domain.CreateInstanceAndUnwrap(
    typeof(Example).Assembly.FullName,
    "PrimeNumberGenerator", true,
    BindingFlags.Default, null,
    new object[] { 1000000 }, null,
    null);
try
{
    start = 100;
    Console.WriteLine(gen.GetPrimesFrom(start));
}
catch (NotPrimeException e)
{
    Console.WriteLine("{0} is not prime", e.NonPrime);
    Console.WriteLine(e);
    Console.WriteLine("-----");
}
}
}

```

Examples

The following example demonstrates a `catch` (with in F#) block that is defined to handle `ArithemticException` errors. This `catch` block also catches `DivideByZeroException` errors, because `DivideByZeroException` derives from `ArithemticException` and there is no `catch` block explicitly defined for `DivideByZeroException` errors.

C#

```
using System;
```

```
class ExceptionTestClass
{
    public static void Main()
    {
        int x = 0;
        try
        {
            int y = 100 / x;
        }
        catch (ArithmeticalException e)
        {
            Console.WriteLine($"ArithmeticalException Handler: {e}");
        }
        catch (Exception e)
        {
            Console.WriteLine($"Generic Exception Handler: {e}");
        }
    }
}
/*
This code example produces the following results:

ArithmeticalException Handler: System.DivideByZeroException: Attempted to
divide by zero.
    at ExceptionTestClass.Main()

*/

```

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

.NET is an open source project.
Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

System.InvalidCastException class

Article • 01/04/2024

This article provides supplementary remarks to the reference documentation for this API.

.NET supports automatic conversion from derived types to their base types and back to the derived type, as well as from types that present interfaces to interface objects and back. It also includes a variety of mechanisms that support custom conversions. For more information, see [Type Conversion in .NET](#).

An [InvalidCastException](#) exception is thrown when the conversion of an instance of one type to another type is not supported. For example, attempting to convert a [Char](#) value to a [DateTime](#) value throws an [InvalidCastException](#) exception. It differs from an [OverflowException](#) exception, which is thrown when a conversion of one type to another is supported, but the value of the source type is outside the range of the target type. An [InvalidCastException](#) exception is caused by developer error and should not be handled in a `try/catch` block. Instead, the cause of the exception should be eliminated.

For information about conversions supported by the system, see the [Convert](#) class. For errors that occur when the destination type can store source type values but is not large enough to store a specific source value, see the [OverflowException](#) exception.

ⓘ Note

In many cases, your language compiler detects that no conversion exists between the source type and the target type and issues a compiler error.

Some of the conditions under which an attempted conversion throws an [InvalidCastException](#) exception are discussed in the following sections.

For an explicit reference conversion to be successful, the source value must be `null`, or the object type referenced by the source argument must be convertible to the destination type by an implicit reference conversion.

The following intermediate language (IL) instructions throw an [InvalidCastException](#) exception:

- `castclass`
- `refanyval`
- `unbox`

[InvalidOperationException](#) uses the HRESULT `COR_E_INVALIDCAST`, which has the value 0x80004002.

For a list of initial property values for an instance of [InvalidOperationException](#), see the [InvalidOperationException](#) constructors.

Primitive types and [IConvertible](#)

You directly or indirectly call a primitive type's [IConvertible](#) implementation that does not support a particular conversion. For example, trying to convert a [Boolean](#) value to a [Char](#) or a [DateTime](#) value to an [Int32](#) throws an [InvalidOperationException](#) exception. The following example calls both the [Boolean.IConvertible.ToChar](#) and [Convert.ToString\(Boolean\)](#) methods to convert a [Boolean](#) value to a [Char](#). In both cases, the method call throws an [InvalidOperationException](#) exception.

C#

```
using System;

public class IConvertibleEx
{
    public static void Main()
    {
        bool flag = true;
        try
        {
            IConvertible conv = flag;
            Char ch = conv.ToChar(null);
            Console.WriteLine("Conversion succeeded.");
        }
        catch (InvalidOperationException)
        {
            Console.WriteLine("Cannot convert a Boolean to a Char.");
        }

        try
        {
            Char ch = Convert.ToChar(flag);
            Console.WriteLine("Conversion succeeded.");
        }
        catch (InvalidOperationException)
        {
            Console.WriteLine("Cannot convert a Boolean to a Char.");
        }
    }
}

// The example displays the following output:
```

```
//      Cannot convert a Boolean to a Char.  
//      Cannot convert a Boolean to a Char.
```

Because the conversion is not supported, there is no workaround.

The Convert.ChangeType method

You've called the [Convert.ChangeType](#) method to convert an object from one type to another, but one or both types don't implement the [IConvertible](#) interface.

In most cases, because the conversion is not supported, there is no workaround. In some cases, a possible workaround is to manually assign property values from the source type to similar properties of the target type.

Narrowing conversions and IConvertible implementations

Narrowing operators define the explicit conversions supported by a type. A casting operator in C# or the `CType` conversion method in Visual Basic (if `Option Strict` is on) is required to perform the conversion.

However, if neither the source type nor the target type defines an explicit or narrowing conversion between the two types, and the [IConvertible](#) implementation of one or both types doesn't support a conversion from the source type to the target type, an [InvalidOperationException](#) exception is thrown.

In most cases, because the conversion is not supported, there is no workaround.

Downcasting

You're downcasting, that is, trying to convert an instance of a base type to one of its derived types. In the following example, trying to convert a `Person` object to a `PersonWithID` object fails.

C#

```
using System;  
  
public class Person  
{  
    String _name;
```

```
public String Name
{
    get { return _name; }
    set { _name = value; }
}

public class PersonWithId : Person
{
    String _id;

    public string Id
    {
        get { return _id; }
        set { _id = value; }
    }
}

public class Example
{
    public static void Main()
    {
        Person p = new Person();
        p.Name = "John";
        try {
            PersonWithId pid = (PersonWithId) p;
            Console.WriteLine("Conversion succeeded.");
        }
        catch (InvalidCastException) {
            Console.WriteLine("Conversion failed.");
        }

        PersonWithId pid1 = new PersonWithId();
        pid1.Name = "John";
        pid1.Id = "246";
        Person p1 = pid1;
        try {
            PersonWithId pid1a = (PersonWithId) p1;
            Console.WriteLine("Conversion succeeded.");
        }
        catch (InvalidCastException) {
            Console.WriteLine("Conversion failed.");
        }

        Person p2 = null;
        try {
            PersonWithId pid2 = (PersonWithId) p2;
            Console.WriteLine("Conversion succeeded.");
        }
        catch (InvalidCastException) {
            Console.WriteLine("Conversion failed.");
        }
    }
}

// The example displays the following output:
```

```
//      Conversion failed.  
//      Conversion succeeded.  
//      Conversion succeeded.
```

As the example shows, the downcast succeeds only if the `Person` object was created by an upcast from a `PersonWithId` object to a `Person` object, or if the `Person` object is `null`.

Conversion from an interface object

You're attempting to convert an interface object to a type that implements that interface, but the target type is not the same type or a base class of the type from which the interface object was originally derived. The following example throws an `InvalidOperationException` exception when it attempts to convert an `IFormatProvider` object to a `DateTimeFormatInfo` object. The conversion fails because although the `DateTimeFormatInfo` class implements the `IFormatProvider` interface, the `DateTimeFormatInfo` object is not related to the `CultureInfo` class from which the interface object was derived.

C#

```
using System;  
using System.Globalization;  
  
public class InterfaceEx  
{  
    public static void Main()  
    {  
        var culture = CultureInfo.InvariantCulture;  
        IFormatProvider provider = culture;  
  
        DateTimeFormatInfo dt = (DateTimeFormatInfo)provider;  
    }  
}  
// The example displays the following output:  
//     Unhandled Exception: System.InvalidCastException:  
//         Unable to cast object of type //System.Globalization.CultureInfo//  
//             to  
//                 type //System.Globalization.DateTimeFormatInfo//.  
//             at Example.Main()
```

As the exception message indicates, the conversion would succeed only if the interface object is converted back to an instance of the original type, in this case a `CultureInfo`. The conversion would also succeed if the interface object is converted to an instance of a base type of the original type.

String conversions

You're trying to convert a value or an object to its string representation by using a casting operator in C#. In the following example, both the attempt to cast a `Char` value to a string and the attempt to cast an integer to a string throw an `InvalidCastException` exception.

C#

```
public class StringEx
{
    public static void Main()
    {
        object value = 12;
        // Cast throws an InvalidCastException exception.
        string s = (string)value;
    }
}
```

① Note

Using the Visual Basic `cstr` operator to convert a value of a primitive type to a string succeeds. The operation does not throw an `InvalidCastException` exception.

To successfully convert an instance of any type to its string representation, call its `ToString` method, as the following example does. The `ToString` method is always present, since the `ToString` method is defined by the `Object` class and therefore is either inherited or overridden by all managed types.

C#

```
using System;

public class ToStringEx2
{
    public static void Main()
    {
        object value = 12;
        string s = value.ToString();
        Console.WriteLine(s);
    }
}
// The example displays the following output:
//      12
```

Visual Basic 6.0 migration

You're upgrading a Visual Basic 6.0 application with a call to a custom event in a user control to Visual Basic .NET, and an `InvalidOperationException` exception is thrown with the message, "Specified cast is not valid." To eliminate this exception, change the line of code in your form (such as `Form1`)

VB

```
Call UserControl11_MyCustomEvent(UserControl11, New  
UserControl1.MyCustomEventArgs(5))
```

and replace it with the following line of code:

VB

```
Call UserControl11_MyCustomEvent(UserControl11(0), New  
UserControl1.MyCustomEventArgs(5))
```

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

.NET is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

System.InvalidOperationException class

Article • 01/04/2024

This article provides supplementary remarks to the reference documentation for this API.

[InvalidOperationException](#) is used in cases when the failure to invoke a method is caused by reasons other than invalid arguments. Typically, it's thrown when the state of an object cannot support the method call. For example, an [InvalidOperationException](#) exception is thrown by methods such as:

- [IEnumerator.MoveNext](#) if objects of a collection are modified after the enumerator is created. For more information, see [Changing a collection while iterating it](#).
- [ResourceSet.GetString](#) if the resource set is closed before the method call is made.
- [XContainer.Add](#), if the object or objects to be added would result in an incorrectly structured XML document.
- A method that attempts to manipulate the UI from a thread that is not the main or UI thread.

Important

Because the [InvalidOperationException](#) exception can be thrown in a wide variety of circumstances, it is important to read the exception message returned by the [Message](#) property.

[InvalidOperationException](#) uses the HRESULT `COR_E_INVALIDOPERATION`, which has the value 0x80131509.

For a list of initial property values for an instance of [InvalidOperationException](#), see the [InvalidOperationException](#) constructors.

Common causes of InvalidOperationException exceptions

The following sections show how some common cases in which an [InvalidOperationException](#) exception is thrown in an app. How you handle the issue depends on the specific situation. Most commonly, however, the exception results from developer error, and the [InvalidOperationException](#) exception can be anticipated and avoided.

Updating a UI thread from a non-UI thread

Often, worker threads are used to perform some background work that involves gathering data to be displayed in an application's user interface. However, most GUI (graphical user interface) application frameworks for .NET, such as Windows Forms and Windows Presentation Foundation (WPF), let you access GUI objects only from the thread that creates and manages the UI (the Main or UI thread). An [InvalidOperationException](#) is thrown when you try to access a UI element from a thread other than the UI thread. The text of the exception message is shown in the following table.

[+] Expand table

| Application Type | Message |
|-------------------|--|
| WPF app | The calling thread cannot access this object because a different thread owns it. |
| UWP app | The application called an interface that was marshaled for a different thread. |
| Windows Forms app | Cross-thread operation not valid: Control 'TextBox1' accessed from a thread other than the thread it was created on. |

UI frameworks for .NET implement a *dispatcher* pattern that includes a method to check whether a call to a member of a UI element is being executed on the UI thread, and other methods to schedule the call on the UI thread:

- In WPF apps, call the [Dispatcher.CheckAccess](#) method to determine if a method is running on a non-UI thread. It returns `true` if the method is running on the UI thread and `false` otherwise. Call one of the overloads of the [Dispatcher.Invoke](#) method to schedule the call on the UI thread.
- In UWP apps, check the [CoreDispatcher.HasThreadAccess](#) property to determine if a method is running on a non-UI thread. Call the [CoreDispatcher.RunAsync](#) method to execute a delegate that updates the UI thread.
- In Windows Forms apps, use the [Control.InvokeRequired](#) property to determine if a method is running on a non-UI thread. Call one of the overloads of the [Control.Invoke](#) method to execute a delegate that updates the UI thread.

The following examples illustrate the [InvalidOperationException](#) exception that is thrown when you attempt to update a UI element from a thread other than the thread that created it. Each example requires that you create two controls:

- A text box control named `textBox1`. In a Windows Forms app, you should set its `Multiline` property to `true`.
- A button control named `threadExampleBtn`. The example provides a handler, `ThreadsExampleBtn_Click`, for the button's `Click` event.

In each case, the `threadExampleBtn_Click` event handler calls the `DoSomeWork` method twice. The first call runs synchronously and succeeds. But the second call, because it runs asynchronously on a thread pool thread, attempts to update the UI from a non-UI thread. This results in a `InvalidOperationException` exception.

WPF apps

C#

```
private async void threadExampleBtn_Click(object sender, RoutedEventArgs e)
{
    textBox1.Text = String.Empty;

    textBox1.Text = "Simulating work on UI thread.\n";
    DoSomeWork(20);
    textBox1.Text += "Work completed...\n";

    textBox1.Text += "Simulating work on non-UI thread.\n";
    await Task.Run(() => DoSomeWork(1000));
    textBox1.Text += "Work completed...\n";
}

private async void DoSomeWork(int milliseconds)
{
    // Simulate work.
    await Task.Delay(milliseconds);

    // Report completion.
    var msg = String.Format("Some work completed in {0} ms.\n",
    milliseconds);
    textBox1.Text += msg;
}
```

The following version of the `DoSomeWork` method eliminates the exception in a WPF app.

C#

```
private async void DoSomeWork(int milliseconds)
{
    // Simulate work.
    await Task.Delay(milliseconds);

    // Report completion.
```

```

    bool uiAccess = textBox1.Dispatcher.CheckAccess();
    String msg = String.Format("Some work completed in {0} ms. on {1}UI
thread\n",
                                milliseconds, uiAccess ? String.Empty : "non-
");
    if (uiAccess)
        textBox1.Text += msg;
    else
        textBox1.Dispatcher.Invoke(() => { textBox1.Text += msg; });
}

```

Windows Forms apps

C#

```

List<String> lines = new List<String>();

private async void threadExampleBtn_Click(object sender, EventArgs e)
{
    textBox1.Text = String.Empty;
    lines.Clear();

    lines.Add("Simulating work on UI thread.");
    textBox1.Lines = lines.ToArray();
    DoSomeWork(20);

    lines.Add("Simulating work on non-UI thread.");
    textBox1.Lines = lines.ToArray();
    await Task.Run(() => DoSomeWork(1000));

    lines.Add("ThreadsExampleBtn_Click completes. ");
    textBox1.Lines = lines.ToArray();
}

private async void DoSomeWork(int milliseconds)
{
    // simulate work
    await Task.Delay(milliseconds);

    // report completion
    lines.Add(String.Format("Some work completed in {0} ms on UI thread.",
milliseconds));
    textBox1.Lines = lines.ToArray();
}

```

The following version of the `DoSomeWork` method eliminates the exception in a Windows Forms app.

C#

```

private async void DoSomeWork(int milliseconds)
{
    // simulate work
    await Task.Delay(milliseconds);

    // Report completion.
    bool uiMarshal = textBox1.InvokeRequired;
    String msg = String.Format("Some work completed in {0} ms. on {1}UI
thread\n",
                                milliseconds, uiMarshal ? String.Empty :
"non-");
    lines.Add(msg);

    if (uiMarshal) {
        textBox1.Invoke(new Action(() => { textBox1.Lines = lines.ToArray();
}));
    }
    else {
        textBox1.Lines = lines.ToArray();
    }
}

```

Changing a collection while iterating it

The `foreach` statement in C#, `for...in` in F#, or `For Each` statement in Visual Basic is used to iterate the members of a collection and to read or modify its individual elements. However, it can't be used to add or remove items from the collection. Doing this throws an `InvalidOperationException` exception with a message that is similar to, "Collection was modified; enumeration operation may not execute."

The following example iterates a collection of integers attempts to add the square of each integer to the collection. The example throws an `InvalidOperationException` with the first call to the `List<T>.Add` method.

C#

```

using System;
using System.Collections.Generic;

public class IteratingEx1
{
    public static void Main()
    {
        var numbers = new List<int>() { 1, 2, 3, 4, 5 };
        foreach (var number in numbers)
        {
            int square = (int)Math.Pow(number, 2);
            Console.WriteLine("{0}^{1}", number, square);
            Console.WriteLine("Adding {0} to the collection...\n", square);
        }
    }
}

```

```

        numbers.Add(square);
    }
}
// The example displays the following output:
//      1^1
//      Adding 1 to the collection...
//
//
//      Unhandled Exception: System.InvalidOperationException: Collection was
modified;
//          enumeration operation may not execute.
//          at
System.ThrowHelper.ThrowInvalidOperationException(ExceptionResource
resource)
//          at System.Collections.Generic.List`1.Enumerator.MoveNextRare()
//          at Example.Main()

```

You can eliminate the exception in one of two ways, depending on your application logic:

- If elements must be added to the collection while iterating it, you can iterate it by index using the `for (for..to in F#)` statement instead of `foreach`, `for...in`, or `For Each`. The following example uses the `for` statement to add the square of numbers in the collection to the collection.

C#

```

using System;
using System.Collections.Generic;

public class IteratingEx2
{
    public static void Main()
    {
        var numbers = new List<int>() { 1, 2, 3, 4, 5 };

        int upperBound = numbers.Count - 1;
        for (int ctr = 0; ctr <= upperBound; ctr++)
        {
            int square = (int)Math.Pow(numbers[ctr], 2);
            Console.WriteLine("{0}^{1}", numbers[ctr], square);
            Console.WriteLine("Adding {0} to the collection...\n",
square);
            numbers.Add(square);
        }

        Console.WriteLine("Elements now in the collection: ");
        foreach (var number in numbers)
            Console.Write("{0}    ", number);
    }
}
```

```

}
// The example displays the following output:
//      1^1
//      Adding 1 to the collection...
//
//      2^4
//      Adding 4 to the collection...
//
//      3^9
//      Adding 9 to the collection...
//
//      4^16
//      Adding 16 to the collection...
//
//      5^25
//      Adding 25 to the collection...
//
//      Elements now in the collection:
//      1      2      3      4      5      1      4      9      16      25

```

Note that you must establish the number of iterations before iterating the collection either by using a counter inside the loop that will exit the loop appropriately, by iterating backward, from `Count - 1` to 0, or, as the example does, by assigning the number of elements in the array to a variable and using it to establish the upper bound of the loop. Otherwise, if an element is added to the collection on every iteration, an endless loop results.

- If it is not necessary to add elements to the collection while iterating it, you can store the elements to be added in a temporary collection that you add when iterating the collection has finished. The following example uses this approach to add the square of numbers in a collection to a temporary collection, and then to combine the collections into a single array object.

C#

```

using System;
using System.Collections.Generic;

public class IteratingEx3
{
    public static void Main()
    {
        var numbers = new List<int>() { 1, 2, 3, 4, 5 };
        var temp = new List<int>();

        // Square each number and store it in a temporary collection.
        foreach (var number in numbers)
        {
            int square = (int)Math.Pow(number, 2);
            temp.Add(square);
        }

        // Print the contents of the temporary collection.
        foreach (var value in temp)
        {
            Console.WriteLine(value);
        }
    }
}

```

```

        }

        // Combine the numbers into a single array.
        int[] combined = new int[numbers.Count + temp.Count];
        Array.Copy(numbers.ToArray(), 0, combined, 0, numbers.Count);
        Array.Copy(temp.ToArray(), 0, combined, numbers.Count,
temp.Count);

        // Iterate the array.
        foreach (var value in combined)
            Console.Write("{0}    ", value);
    }
}

// The example displays the following output:
//      1    2    3    4    5    1    4    9    16    25

```

Sorting an array or collection whose objects cannot be compared

General-purpose sorting methods, such as the [Array.Sort\(Array\)](#) method or the [List<T>.Sort\(\)](#) method, usually require that at least one of the objects to be sorted implement the [IComparable<T>](#) or the [IComparable](#) interface. If not, the collection or array cannot be sorted, and the method throws an [InvalidOperationException](#) exception. The following example defines a `Person` class, stores two `Person` objects in a generic `List<T>` object, and attempts to sort them. As the output from the example shows, the call to the [List<T>.Sort\(\)](#) method throws an [InvalidOperationException](#).

C#

```

using System;
using System.Collections.Generic;

public class Person1
{
    public Person1(string fName, string lName)
    {
        FirstName = fName;
        LastName = lName;
    }

    public string FirstName { get; set; }
    public string LastName { get; set; }
}

public class ListSortEx1
{
    public static void Main()
    {
        var people = new List<Person1>();

```

```

        people.Add(new Person1("John", "Doe"));
        people.Add(new Person1("Jane", "Doe"));
        people.Sort();
        foreach (var person in people)
            Console.WriteLine("{0} {1}", person.FirstName, person.LastName);
    }
}

// The example displays the following output:
//      Unhandled Exception: System.InvalidOperationException: Failed to
//      compare two elements in the array. --->
//          System.ArgumentException: At least one object must implement
// IComparable.
//          at System.Collections.Comparer.Compare(Object a, Object b)
//          at System.Collections.Generic.ArraySortHelper`1.SwapIfGreater(T[]
keys, IComparer`1 comparer, Int32 a, Int32 b)
//          at
System.Collections.Generic.ArraySortHelper`1.DepthLimitedQuickSort(T[] keys,
Int32 left, Int32 right, IComparer`1 comparer, Int32 depthLimit)
//          at System.Collections.Generic.ArraySortHelper`1.Sort(T[] keys,
Int32 index, Int32 length, IComparer`1 comparer)
//          --- End of inner exception stack trace ---
//          at System.Collections.Generic.ArraySortHelper`1.Sort(T[] keys,
Int32 index, Int32 length, IComparer`1 comparer)
//          at System.Array.Sort[T](T[] array, Int32 index, Int32 length,
IComparer`1 comparer)
//          at System.Collections.Generic.List`1.Sort(Int32 index, Int32 count,
IComparer`1 comparer)
//          at Example.Main()

```

You can eliminate the exception in any of three ways:

- If you can own the type that you are trying to sort (that is, if you control its source code), you can modify it to implement the `IComparable<T>` or the `IComparable` interface. This requires that you implement either the `IComparable<T>.CompareTo` or the `CompareTo` method. Adding an interface implementation to an existing type is not a breaking change.

The following example uses this approach to provide an `IComparable<T>` implementation for the `Person` class. You can still call the collection or array's general sorting method and, as the output from the example shows, the collection sorts successfully.

C#

```

using System;
using System.Collections.Generic;

public class Person2 : IComparable<Person>
{

```

```

public Person2(String fName, String lName)
{
    FirstName = fName;
    LastName = lName;
}

public String FirstName { get; set; }
public String LastName { get; set; }

public int CompareTo(Person other)
{
    return String.Format("{0} {1}", LastName, FirstName).
        CompareTo(String.Format("{0} {1}", other.LastName,
other.FirstName));
}
}

public class ListSortEx2
{
    public static void Main()
    {
        var people = new List<Person2>();

        people.Add(new Person2("John", "Doe"));
        people.Add(new Person2("Jane", "Doe"));
        people.Sort();
        foreach (var person in people)
            Console.WriteLine("{0} {1}", person.FirstName,
person.LastName);
    }
}
// The example displays the following output:
//      Jane Doe
//      John Doe

```

- If you cannot modify the source code for the type you are trying to sort, you can define a special-purpose sorting class that implements the `IComparer<T>` interface. You can call an overload of the `Sort` method that includes an `IComparer<T>` parameter. This approach is especially useful if you want to develop a specialized sorting class that can sort objects based on multiple criteria.

The following example uses the approach by developing a custom `PersonComparer` class that is used to sort `Person` collections. It then passes an instance of this class to the `List<T>.Sort(IComparer<T>)` method.

C#

```

using System;
using System.Collections.Generic;

public class Person3

```

```

{
    public Person3(String fName, String lName)
    {
        FirstName = fName;
        LastName = lName;
    }

    public String FirstName { get; set; }
    public String LastName { get; set; }
}

public class PersonComparer : IComparer<Person3>
{
    public int Compare(Person3 x, Person3 y)
    {
        return String.Format("{0} {1}", x.LastName, x.FirstName).
            CompareTo(String.Format("{0} {1}", y.LastName,
y.FirstName));
    }
}

public class ListSortEx3
{
    public static void Main()
    {
        var people = new List<Person3>();

        people.Add(new Person3("John", "Doe"));
        people.Add(new Person3("Jane", "Doe"));
        people.Sort(new PersonComparer());
        foreach (var person in people)
            Console.WriteLine("{0} {1}", person.FirstName,
person.LastName);
    }
}
// The example displays the following output:
//      Jane Doe
//      John Doe

```

- If you cannot modify the source code for the type you are trying to sort, you can create a `Comparison<T>` delegate to perform the sorting. The delegate signature is

C#

```
int Comparison<T>(T x, T y)
```

The following example uses the approach by defining a `PersonComparison` method that matches the `Comparison<T>` delegate signature. It then passes this delegate to the `List<T>.Sort(Comparison<T>)` method.

C#

```
using System;
using System.Collections.Generic;

public class Person
{
    public Person(String fName, String lName)
    {
        FirstName = fName;
        LastName = lName;
    }

    public String FirstName { get; set; }
    public String LastName { get; set; }
}

public class ListSortEx4
{
    public static void Main()
    {
        var people = new List<Person>();

        people.Add(new Person("John", "Doe"));
        people.Add(new Person("Jane", "Doe"));
        people.Sort(PersonComparison);
        foreach (var person in people)
            Console.WriteLine("{0} {1}", person.FirstName,
person.LastName);
    }

    public static int PersonComparison(Person x, Person y)
    {
        return String.Format("{0} {1}", x.LastName, x.FirstName).
            CompareTo(String.Format("{0} {1}", y.LastName,
y.FirstName));
    }
}
// The example displays the following output:
//      Jane Doe
//      John Doe
```

Casting a Nullable<T> that's null to its underlying type

Attempting to cast a `Nullable<T>` value that is `null` to its underlying type throws an `InvalidOperationException` exception and displays the error message, **"Nullable object must have a value.**

The following example throws an `InvalidOperationException` exception when it attempts to iterate an array that includes a `Nullable(Of Integer)` value.

C#

```
using System;
using System.Linq;

public class NullableEx1
{
    public static void Main()
    {
        var queryResult = new int?[] { 1, 2, null, 4 };
        var map = queryResult.Select(nullableInt => (int)nullableInt);

        // Display list.
        foreach (var num in map)
            Console.Write("{0} ", num);
        Console.WriteLine();
    }
}

// The example displays the following output:
//      1 2
//      Unhandled Exception: System.InvalidOperationException: Nullable object
//      must have a value.
//          at
System.ThrowHelper.ThrowInvalidOperationException(ExceptionResource
resource)
//          at Example.<Main>b__0(Nullable`1 nullableInt)
//          at System.Linq.Enumerable.WhereSelectArrayIterator`2.MoveNext()
//          at Example.Main()
```

To prevent the exception:

- Use the `Nullable<T>.HasValue` property to select only those elements that are not `null`.
- Call one of the `Nullable<T>.GetValueOrDefault` overloads to provide a default value for a `null` value.

The following example does both to avoid the `InvalidOperationException` exception.

C#

```
using System;
using System.Linq;

public class NullableEx2
{
    public static void Main()
    {
        var queryResult = new int?[] { 1, 2, null, 4 };
        var numbers = queryResult.Select(nullableInt =>
(int)nullableInt.GetValueOrDefault());
```

```

    // Display list using Nullable<int>.HasValue.
    foreach (var number in numbers)
        Console.Write("{0} ", number);
    Console.WriteLine();

    numbers = queryResult.Select(nullableInt => (int)
(nullableInt.HasValue ? nullableInt : -1));
    // Display list using Nullable<int>.GetValueOrDefault.
    foreach (var number in numbers)
        Console.Write("{0} ", number);
    Console.WriteLine();
}

// The example displays the following output:
//      1 2 0 4
//      1 2 -1 4

```

Call a System.Linq.Enumerable method on an empty collection

The `Enumerable.Average`, `Enumerable.First`, `Enumerable.Last`, `Enumerable.Max`, `Enumerable.Min`, `Enumerable.Single`, and `Enumerable.SingleOrDefault` methods perform operations on a sequence and return a single result. Some overloads of these methods throw an `InvalidOperationException` exception when the sequence is empty, while other overloads return `null`. The `Enumerable.SingleOrDefault` method also throws an `InvalidOperationException` exception when the sequence contains more than one element.

Note

Most of the methods that throw an `InvalidOperationException` exception are overloads. Be sure that you understand the behavior of the overload that you choose.

The following table lists the exception messages from the `InvalidOperationException` exception objects thrown by calls to some `System.Linq.Enumerable` methods.

 Expand table

| Method | Message |
|------------------------|-------------------------------|
| <code>Aggregate</code> | Sequence contains no elements |
| <code>Average</code> | |
| <code>Last</code> | |

| Method | Message |
|-----------------|--|
| Max | |
| Min | |
| First | Sequence contains no matching element |
| Single | Sequence contains more than one matching element |
| SingleOrDefault | |

How you eliminate or handle the exception depends on your application's assumptions and on the particular method you call.

- When you deliberately call one of these methods without checking for an empty sequence, you are assuming that the sequence is not empty, and that an empty sequence is an unexpected occurrence. In this case, catching or rethrowing the exception is appropriate.
- If your failure to check for an empty sequence was inadvertent, you can call one of the overloads of the [Enumerable.Any](#) overload to determine whether a sequence contains any elements.

Tip

Calling the [Enumerable.Any<TSource>\(IQueryable<TSource>, Func<TSource, Boolean>\)](#) method before generating a sequence can improve performance if the data to be processed might contain a large number of elements or if operation that generates the sequence is expensive.

- If you've called a method such as [Enumerable.First](#), [Enumerable.Last](#), or [Enumerable.Single](#), you can substitute an alternate method, such as [Enumerable.FirstOrDefault](#), [Enumerable.LastOrDefault](#), or [Enumerable.SingleOrDefault](#), that returns a default value instead of a member of the sequence.

The examples provide additional detail.

The following example uses the [Enumerable.Average](#) method to compute the average of a sequence whose values are greater than 4. Since no values from the original array exceed 4, no values are included in the sequence, and the method throws an [InvalidOperationException](#) exception.

C#

```

using System;
using System.Linq;

public class Example
{
    public static void Main()
    {
        int[] data = { 1, 2, 3, 4 };
        var average = data.Where(num => num > 4).Average();
        Console.WriteLine("The average of numbers greater than 4 is {0}",
                        average);
    }
}
// The example displays the following output:
//      Unhandled Exception: System.InvalidOperationException: Sequence
contains no elements
//          at System.Linq.Enumerable.Average(IEnumerable`1 source)
//          at Example.Main()

```

The exception can be eliminated by calling the [Any](#) method to determine whether the sequence contains any elements before calling the method that processes the sequence, as the following example shows.

C#

```

using System;
using System.Linq;

public class EnumerableEx2
{
    public static void Main()
    {
        int[] dbQueryResults = { 1, 2, 3, 4 };
        var moreThan4 = dbQueryResults.Where(num => num > 4);

        if (moreThan4.Any())
            Console.WriteLine("Average value of numbers greater than 4:
{0}:",
                            moreThan4.Average());
        else
            // handle empty collection
            Console.WriteLine("The dataset has no values greater than 4.");
    }
}
// The example displays the following output:
//      The dataset has no values greater than 4.

```

The [Enumerable.First](#) method returns the first item in a sequence or the first element in a sequence that satisfies a specified condition. If the sequence is empty and therefore does not have a first element, it throws an [InvalidOperationException](#) exception.

In the following example, the `Enumerable.First<TSource>(IEnumerable<TSource>, Func<TSource, Boolean>)` method throws an `InvalidOperationException` exception because the `dbQueryResults` array doesn't contain an element greater than 4.

C#

```
using System;
using System.Linq;

public class EnumerableEx3
{
    public static void Main()
    {
        int[] dbQueryResults = { 1, 2, 3, 4 };

        var firstNum = dbQueryResults.First(n => n > 4);

        Console.WriteLine("The first value greater than 4 is {0}",
                          firstNum);
    }
}

// The example displays the following output:
//     Unhandled Exception: System.InvalidOperationException:
//         Sequence contains no matching element
//             at System.Linq.Enumerable.First[TSource](IEnumerable`1 source,
Func`2 predicate)
//             at Example.Main()
```

You can call the `Enumerable.FirstOrDefault` method instead of `Enumerable.First` to return a specified or default value. If the method does not find a first element in the sequence, it returns the default value for that data type. The default value is `null` for a reference type, zero for a numeric data type, and `DateTime.MinValue` for the `DateTime` type.

ⓘ Note

Interpreting the value returned by the `Enumerable.FirstOrDefault` method is often complicated by the fact that the default value of the type can be a valid value in the sequence. In this case, you can call the `Enumerable.Any` method to determine whether the sequence has valid members before calling the `Enumerable.First` method.

The following example calls the `Enumerable.FirstOrDefault<TSource>(IEnumerable<TSource>, Func<TSource, Boolean>)` method to prevent the `InvalidOperationException` exception thrown in the previous example.

C#

```

using System;
using System.Linq;

public class EnumerableEx4
{
    public static void Main()
    {
        int[] dbQueryResults = { 1, 2, 3, 4 };

        var firstNum = dbQueryResults.FirstOrDefault(n => n > 4);

        if (firstNum == 0)
            Console.WriteLine("No value is greater than 4.");
        else
            Console.WriteLine("The first value greater than 4 is {0}",
                firstNum);
    }
}

// The example displays the following output:
//      No value is greater than 4.

```

Call `Enumerable.Single` or `Enumerable.SingleOrDefault` on a sequence without one element

The `Enumerable.Single` method returns the only element of a sequence, or the only element of a sequence that meets a specified condition. If there are no elements in the sequence, or if there is more than one element , the method throws an `InvalidOperationException` exception.

You can use the `Enumerable.SingleOrDefault` method to return a default value instead of throwing an exception when the sequence contains no elements. However, the `Enumerable.SingleOrDefault` method still throws an `InvalidOperationException` exception when the sequence contains more than one element.

The following table lists the exception messages from the `InvalidOperationException` exception objects thrown by calls to the `Enumerable.Single` and `Enumerable.SingleOrDefault` methods.

[+] Expand table

| Method | Message |
|-----------------|--|
| Single | Sequence contains no matching element |
| Single | Sequence contains more than one matching element |
| SingleOrDefault | |

In the following example, the call to the `Enumerable.Single` method throws an `InvalidOperationException` exception because the sequence doesn't have an element greater than 4.

C#

```
using System;
using System.Linq;

public class EnumerableEx5
{
    public static void Main()
    {
        int[] dbQueryResults = { 1, 2, 3, 4 };

        var singleObject = dbQueryResults.Single(value => value > 4);

        // Display results.
        Console.WriteLine("{0} is the only value greater than 4",
singleObject);
    }
}

// The example displays the following output:
//     Unhandled Exception: System.InvalidOperationException:
//         Sequence contains no matching element
//             at System.Linq.Enumerable.Single[TSource](IEnumerable`1 source,
Func`2 predicate)
//             at Example.Main()
```

The following example attempts to prevent the `InvalidOperationException` exception thrown when a sequence is empty by instead calling the `Enumerable.SingleOrDefault` method. However, because this sequence returns multiple elements whose value is greater than 2, it also throws an `InvalidOperationException` exception.

C#

```
using System;
using System.Linq;

public class EnumerableEx6
{
    public static void Main()
    {
        int[] dbQueryResults = { 1, 2, 3, 4 };

        var singleObject = dbQueryResults.SingleOrDefault(value => value >
2);

        if (singleObject != 0)
            Console.WriteLine("{0} is the only value greater than 2",
singleObject);
```

```
        else
            // Handle an empty collection.
            Console.WriteLine("No value is greater than 2");
    }
}

// The example displays the following output:
//      Unhandled Exception: System.InvalidOperationException:
//          Sequence contains more than one matching element
//          at System.Linq.Enumerable.SingleOrDefault[TSource](IEnumerable`1
source, Func`2 predicate)
//          at Example.Main()
```

Calling the [Enumerable.Single](#) method assumes that either a sequence or the sequence that meets specified criteria contains only one element. [Enumerable.SingleOrDefault](#) assumes a sequence with zero or one result, but no more. If this assumption is a deliberate one on your part and these conditions are not met, rethrowing or catching the resulting [InvalidOperationException](#) is appropriate. Otherwise, or if you expect that invalid conditions will occur with some frequency, you should consider using some other [Enumerable](#) method, such as [FirstOrDefault](#) or [Where](#).

Dynamic cross-application domain field access

The [OpCodes.Ldflda](#) Microsoft intermediate language (MSIL) instruction throws an [InvalidOperationException](#) exception if the object containing the field whose address you are trying to retrieve is not within the application domain in which your code is executing. The address of a field can only be accessed from the application domain in which it resides.

Throw an [InvalidOperationException](#) exception

You should throw an [InvalidOperationException](#) exception only when the state of your object for some reason does not support a particular method call. That is, the method call is valid in some circumstances or contexts, but is invalid in others.

If the method invocation failure is due to invalid arguments, then [ArgumentException](#) or one of its derived classes, [ArgumentNullException](#) or [ArgumentOutOfRangeException](#), should be thrown instead.

 Collaborate with us on
GitHub



.NET feedback

.NET is an open source project.
Select a link to provide feedback:

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

 [Open a documentation issue](#)

 [Provide product feedback](#)

System.NotImplementedException class

Article • 01/04/2024

This article provides supplementary remarks to the reference documentation for this API.

The [NotImplementedException](#) exception is thrown when a particular method, get accessor, or set accessor is present as a member of a type but is not implemented.

[NotImplementedException](#) uses the default [Object.Equals](#) implementation, which supports reference equality. For a list of initial values for an instance of [NotImplementedException](#), see the [NotImplementedException](#) constructors.

Throw the exception

You might choose to throw a [NotImplementedException](#) exception in properties or methods in your own types when the that member is still in development and will only later be implemented in production code. In other words, a [NotImplementedException](#) exception should be synonymous with "still in development."

Handle the exception

The [NotImplementedException](#) exception indicates that the method or property that you are attempting to invoke has no implementation and therefore provides no functionality. As a result, you should not handle this error in a `try/catch` block. Instead, you should remove the member invocation from your code. You can include a call to the member when it is implemented in the production version of a library.

In some cases, a [NotImplementedException](#) exception may not be used to indicate functionality that is still in development in a pre-production library. However, this still indicates that the functionality is unavailable, and you should remove the member invocation from your code.

NotImplementedException and other exception types

.NET also includes two other exception types, [NotSupportedException](#) and [PlatformNotSupportedException](#), that indicate that no implementation exists for a

particular member of a type. You should throw one of these instead of a [NotImplementedException](#) exception under the following conditions:

- Throw a [PlatformNotSupportedException](#) exception on platforms on which the functionality is not supported if you've designed a type with one or more members that are available on some platforms or versions but not others.
- Throw a [NotSupportedException](#) exception if the implementation of an interface member or an override to an abstract base class method is not possible.

For example, the [Convert.ToInt32\(DateTime\)](#) method throws a [NotSupportedException](#) exception because no meaningful conversion between a date and time and a 32-bit signed integer exists. The method must be present in this case because the [Convert](#) class implements the [IConvertible](#) interface.

You should also throw a [NotSupportedException](#) exception if you've implemented an abstract base class and add a new member to it that must be overridden by derived classes. In that case, making the member abstract causes existing subclasses to fail to load.

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



.NET feedback

.NET is an open source project.
Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

System.NotSupportedException class

Article • 01/04/2024

This article provides supplementary remarks to the reference documentation for this API.

[NotSupportedException](#) indicates that no implementation exists for an invoked method or property.

[NotSupportedException](#) uses the HRESULT `COR_E_NOTSUPPORTED`, which has the value 0x80131515.

For a list of initial property values for an instance of [NotSupportedException](#), see the [NotSupportedException](#) constructors.

Throw a NotSupportedException exception

You might consider throwing a [NotSupportedException](#) exception in the following cases:

- You're implementing a general-purpose interface, and number of the methods have no meaningful implementation. For example, if you are creating a date and time type that implements the [IConvertible](#) interface, you would throw a [NotSupportedException](#) exception for most of the conversions.
- You've inherited from an abstract class that requires that you override a number of methods. However, you're only prepared to provide an implementation for a subset of these. For the methods that you decide not to implement, you can choose to throw a [NotSupportedException](#).
- You're defining a general-purpose type with a state that enables operations conditionally. For example, your type can be either read-only or read-write. In that case:
 - If the object is read-only, attempting to assign values to the properties of an instance or call methods that modify instance state should throw a [NotSupportedException](#) exception.
 - You should implement a property that returns a [Boolean](#) value that indicates whether particular functionality is available. For example, for a type that can be either read-only or read-write, you could implement a `IsReadOnly` property that indicates whether the set of read-write methods are available or unavailable.

Handle a NotSupportedException exception

The [NotSupportedException](#) exception indicates that a method has no implementation and that you should not call it. You should not handle the exception. Instead, what you should do depends on the cause of the exception: whether an implementation is completely absent, or the member invocation is inconsistent with the purpose of an object (such as a call to the [FileStream.Write](#) method on a read-only [FileStream](#) object).

An implementation has not been provided because the operation cannot be performed in a meaningful way. This is a common exception when you are calling methods on an object that provides implementations for the methods of an abstract base class, or that implements a general-purpose interface, and the method has no meaningful implementation.

For example, the [Convert](#) class implements the [IConvertible](#) interface, which means that it must include a method to convert every primitive type to every other primitive type. Many of those conversions, however, are not possible. As a result, a call to the [Convert.ToBoolean\(DateTime\)](#) method, for instance, throws a [NotSupportedException](#) exception because there is no possible conversion between a [DateTime](#) and a [Boolean](#) value

To eliminate the exception, you should eliminate the method call.

The method call is not supported given the state of the object. You're attempting to invoke a member whose functionality is unavailable because of the object's state. You can eliminate the exception in one of three ways:

- You know the state of the object in advance, but you've invoked an unsupported method or property. In this case, the member invocation is an error, and you can eliminate it.
- You know the state of the object in advance (usually because your code has instantiated it), but the object is mis-configured. The following example illustrates this issue. It creates a read-only [FileStream](#) object and then attempts to write to it.

C#

```
using System;
using System.IO;
using System.Text;
using System.Threading.Tasks;

public class Example
{
    public static async Task Main()
```

```

    {
        Encoding enc = Encoding.Unicode;
        String value = "This is a string to persist.";
        Byte[] bytes = enc.GetBytes(value);

        FileStream fs = new FileStream(@".\TestFile.dat",
                                         FileMode.Open,
                                         FileAccess.Read);
        Task t = fs.WriteAsync(enc.GetPreamble(), 0,
                               enc.GetPreamble().Length);
        Task t2 = t.ContinueWith((a) => fs.WriteAsync(bytes, 0,
                                                       bytes.Length));
        await t2;
        fs.Close();
    }
}

// The example displays the following output:
// Unhandled Exception: System.NotSupportedException: Stream does
// not support writing.
//     at System.IO.Stream.BeginWriteInternal(Byte[] buffer, Int32
// offset, Int32 count, AsyncCallback callback, Object state
// , Boolean serializeAsynchronously)
//     at System.IO.FileStream.BeginWrite(Byte[] array, Int32 offset,
// Int32 numBytes, AsyncCallback userCallback, Object sta
// teObject)
//     at System.IO.Stream.<>c.<BeginEndWriteAsync>b__53_0(Stream
// stream, ReadWriteParameters args, AsyncCallback callback,
// Object state)
//     at
System.Threading.Tasks.TaskFactory`1.FromAsyncTrim[TInstance, TArgs]
(TInstance thisRef, TArgs args, Func`5 beginMet
// hod, Func`3 endMethod)
//     at System.IO.Stream.BeginEndWriteAsync(Byte[] buffer, Int32
offset, Int32 count)
//     at System.IO.FileStream.WriteAsync(Byte[] buffer, Int32
offset, Int32 count, CancellationToken cancellationToken)
//     at System.IO.Stream.WriteAsync(Byte[] buffer, Int32 offset,
Int32 count)
//     at Example.Main()

```

You can eliminate the exception by ensuring that the instantiated object supports the functionality you intend. The following example addresses the problem of the read-only [FileStream](#) object by providing the correct arguments to the [FileStream\(FileStream, FileMode, FileAccess\)](#) constructor.

- You don't know the state of the object in advance, and the object doesn't support a particular operation. In most cases, the object should include a property or method that indicates whether it supports a particular set of operations. You can eliminate the exception by checking the value of the object and invoking the member only if appropriate.

The following example defines a `DetectEncoding` method that throws a `NotSupportedException` exception when it attempts to read from the beginning of a stream that does not support read access.

C#

```
using System;
using System.IO;
using System.Threading.Tasks;

public class TestPropEx1
{
    public static async Task Main()
    {
        String name = @"\TestFile.dat";
        var fs = new FileStream(name,
                               FileMode.Create,
                               FileAccess.Write);
        Console.WriteLine("Filename: {0}, Encoding: {1}",
                          name, await
        FileUtilities1.GetEncodingType(fs));
    }
}

public class FileUtilities1
{
    public enum EncodingType
    { None = 0, Unknown = -1, Utf8 = 1, Utf16 = 2, Utf32 = 3 }

    public async static Task<EncodingType> GetEncodingType(FileStream
fs)
    {
        Byte[] bytes = new Byte[4];
        int bytesRead = await fs.ReadAsync(bytes, 0, 4);
        if (bytesRead < 2)
            return EncodingType.None;

        if (bytesRead >= 3 & (bytes[0] == 0xEF && bytes[1] == 0xBB &&
bytes[2] == 0xBF))
            return EncodingType.Utf8;

        if (bytesRead == 4)
        {
            var value = BitConverter.ToInt32(bytes, 0);
            if (value == 0x0000FEFF | value == 0xFEFF0000)
                return EncodingType.Utf32;
        }

        var value16 = BitConverter.ToInt16(bytes, 0);
        if (value16 == (ushort)0xFEFF | value16 == (ushort)0xFFFF)
            return EncodingType.Utf16;

        return EncodingType.Unknown;
    }
}
```

```

    }
// The example displays the following output:
//      Unhandled Exception: System.NotSupportedException: Stream does
//      not support reading.
//          at System.IO.FileStream.BeginRead(Byte[] array, Int32 offset,
//      Int32 numBytes, AsyncCallback callback, Object state)
//          at System.IO.Stream.<>c.<BeginEndReadAsync>b__46_0(Stream
//      stream, ReadWriteParameters args, AsyncCallback callback, Object state)
//          at
System.Threading.Tasks.TaskFactory`1.FromAsyncTrim[TInstance, TArgs]
(TInstance thisRef, TArgs args, Func`5 beginMethod, Func`3 endMethod)
//          at System.IO.Stream.BeginEndReadAsync(Byte[] buffer, Int32
offset, Int32 count)
//          at System.IO.FileStream.ReadAsync(Byte[] buffer, Int32 offset,
Int32 count, CancellationToken cancellationToken)
//          at System.IO.Stream.ReadAsync(Byte[] buffer, Int32 offset,
Int32 count)
//          at FileUtilities.GetEncodingType(FileStream fs) in
C:\Work\docs\program.cs:line 26
//          at Example.Main() in C:\Work\docs\program.cs:line 13
//          at Example.<Main>()

```

You can eliminate the exception by examining the value of the [FileStream.CanRead](#) property and exiting the method if the stream is read-only.

C#

```

public static async Task<EncodingType> GetEncodingType(FileStream
fs)
{
    if (!fs.CanRead)
        return EncodingType.Unknown;

    Byte[] bytes = new Byte[4];
    int bytesRead = await fs.ReadAsync(bytes, 0, 4);
    if (bytesRead < 2)
        return EncodingType.None;

    if (bytesRead >= 3 & (bytes[0] == 0xEF && bytes[1] == 0xBB &&
bytes[2] == 0xBF))
        return EncodingType.Utf8;

    if (bytesRead == 4)
    {
        var value = BitConverter.ToInt32(bytes, 0);
        if (value == 0x0000FEFF | value == 0xFEFF0000)
            return EncodingType.Utf32;
    }

    var value16 = BitConverter.ToInt16(bytes, 0);
    if (value16 == (ushort)0xFEFF | value16 == (ushort)0xFFFF)
        return EncodingType.Utf16;
}

```

```
        return EncodingType.Unknown;
    }
}
// The example displays the following output:
//     Filename: .\TestFile.dat, Encoding: Unknown
```

Related exception types

The [NotSupportedException](#) exception is closely related to two other exception types;

- [NotImplementedException](#)

This exception is thrown when a method could be implemented but is not, either because the member will be implemented in a later version, the member is not available on a particular platform, or the member belongs to an abstract class and a derived class must provide an implementation.

- [InvalidOperationException](#)

This exception is thrown in scenarios in which it is generally sometimes possible for the object to perform the requested operation, and the object state determines whether the operation can be performed.

.NET Compact Framework notes

When working with the .NET Compact Framework and using P/Invoke on a native function, this exception may be thrown if:

- The declaration in managed code is incorrect.
- The .NET Compact Framework does not support what you are trying to do.
- The DLL names are mangled on export.

If a [NotSupportedException](#) exception is thrown, check:

- For any violations of the .NET Compact Framework P/Invoke restrictions.
- For any arguments that require pre-allocated memory. If these exist, you should pass a reference to an existing variable.
- That the names of the exported functions are correct. This can be verified with [DumpBin.exe](#).
- That you are not attempting to pass too many arguments.

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

.NET is an open source project.
Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

System.TypeInitializationException class

Article • 01/04/2024

This article provides supplementary remarks to the reference documentation for this API.

When a class initializer fails to initialize a type, a [TypeInitializationException](#) is created and passed a reference to the exception thrown by the type's class initializer. The [InnerException](#) property of [TypeInitializationException](#) holds the underlying exception.

Typically, the [TypeInitializationException](#) exception reflects a catastrophic condition (the runtime is unable to instantiate a type) that prevents an application from continuing. Most commonly, the [TypeInitializationException](#) is thrown in response to some change in the executing environment of the application. Consequently, other than possibly for troubleshooting debug code, the exception should not be handled in a `try/catch` block. Instead, the cause of the exception should be investigated and eliminated.

[TypeInitializationException](#) uses the HRESULT `COR_E_TYPEINITIALIZATION`, which has the value 0x80131534.

For a list of initial property values for an instance of [TypeInitializationException](#), see the [TypeInitializationException](#) constructors.

The following sections describe some of the situations in which a [TypeInitializationException](#) exception is thrown.

Static constructors

A static constructor, if one exists, is called automatically by the runtime before creating a new instance of a type. Static constructors can be explicitly defined by a developer. If a static constructor is not explicitly defined, compilers automatically create one to initialize any `static` (in C# or F#) or `Shared` (in Visual Basic) members of the type. For more information on static constructors, see [Static Constructors](#).

Most commonly, a [TypeInitializationException](#) exception is thrown when a static constructor is unable to instantiate a type. The [InnerException](#) property indicates why the static constructor was unable to instantiate the type. Some of the more common causes of a [TypeInitializationException](#) exception are:

- An unhandled exception in a static constructor

If an exception is thrown in a static constructor, that exception is wrapped in a [TypeInitializationException](#) exception, and the type cannot be instantiated.

What often makes this exception difficult to troubleshoot is that static constructors are not always explicitly defined in source code. A static constructor exists in a type if:

- It has been explicitly defined as a member of a type.
- The type has `static` (in C# or F#) or `Shared` (in Visual Basic) variables that are declared and initialized in a single statement. In this case, the language compiler generates a static constructor for the type. You can inspect it by using a utility such as [IL Disassembler](#). For instance, when the C# and VB compilers compile the following example, they generate the IL for a static constructor that is similar to this:

```
il

.method private specialname rtspecialname static
    void .cctor() cil managed
{
    // Code size      12 (0xc)
    .maxstack 8
    IL_0000: ldc.i4.3
    IL_0001: newobj     instance void TestClass::ctor(int32)
    IL_0006: stsfld     class TestClass Example::test
    IL_000b: ret
} // end of method Example::cctor
```

The following example shows a [TypeInitializationException](#) exception thrown by a compiler-generated static constructor. The `Example` class includes a `static` (in C#) or `Shared` (in Visual Basic) field of type `TestClass` that is instantiated by passing a value of 3 to its class constructor. That value, however, is illegal; only values of 0 or 1 are permitted. As a result, the `TestClass` class constructor throws an [ArgumentOutOfRangeException](#). Since this exception is not handled, it is wrapped in a [TypeInitializationException](#) exception.

```
C#

using System;

public class Example
{
    private static TestClass test = new TestClass(3);

    public static void Main()
    {
```

```

        Example ex = new Example();
        Console.WriteLine(test.Value);
    }

}

public class TestClass
{
    public readonly int Value;

    public TestClass(int value)
    {
        if (value < 0 || value > 1) throw new
ArgumentOutOfRangeException(nameof(value));
        Value = value;
    }
}

// The example displays the following output:
//     Unhandled Exception: System.TypeInitializationException:
//         The type initializer for 'Example' threw an exception. --->
//         System.ArgumentOutOfRangeException: Specified argument was out
of the range of valid values.
//         at TestClass..ctor(Int32 value)
//         at Example..cctor()
//         --- End of inner exception stack trace ---
//         at Example.Main()

```

Note that the exception message displays information about the [InnerException](#) property.

- A missing assembly or data file

A common cause of a [TypeInitializationException](#) exception is that an assembly or data file that was present in an application's development and test environments is missing from its runtime environment. For example, you can compile the following example to an assembly named Missing1a.dll by using this command-line syntax:

C#

```
csc -t:library Missing1a.cs
```

C#

```

using System;

public class InfoModule
{
    private DateTime firstUse;
    private int ctr = 0;

    public InfoModule(DateTime dat)

```

```
    firstUse = dat;
}

public int Increment()
{
    return ++ctr;
}

public DateTime GetInitializationTime()
{
    return firstUse;
}
}
```

You can then compile the following example to an executable named Missing1.exe by including a reference to Missing1a.dll:

C#

```
csc Missing1.cs /r:Missing1a.dll
```

However, if you rename, move, or delete Missing1a.dll and run the example, it throws a [TypeInitializationException](#) exception and displays the output shown in the example. Note that the exception message includes information about the [InnerException](#) property. In this case, the inner exception is a [FileNotFoundException](#) that is thrown because the runtime cannot find the dependent assembly.

C#

```
using System;

public class MissingEx1
{
    public static void Main()
    {
        Person p = new Person("John", "Doe");
        Console.WriteLine(p);
    }
}

public class Person
{
    static readonly InfoModule s_infoModule;

    readonly string _fName;
    readonly string _lName;

    static Person()
```

```

    {
        s_infoModule = new InfoModule(DateTime.UtcNow);
    }

    public Person(string fName, string lName)
    {
        _fName = fName;
        _lName = lName;
        s_infoModule.Increment();
    }

    public override string ToString()
    {
        return string.Format("{0} {1}", _fName, _lName);
    }
}

// The example displays the following output if missing1a.dll is
// renamed or removed:
//    Unhandled Exception: System.TypeInitializationException:
//        The type initializer for 'Person' threw an exception. --->
//        System.IO.FileNotFoundException: Could not load file or
// assembly
//        'Missing1a, Version=0.0.0.0, Culture=neutral,
// PublicKeyToken=null'
//        or one of its dependencies. The system cannot find the file
// specified.
//        at Person..cctor()
//        --- End of inner exception stack trace ---
//        at Person..ctor(String fName, String lName)
//        at Example.Main()

```

Note

In this example, a `TypeInitializationException` exception was thrown because an assembly could not be loaded. The exception can also be thrown if a static constructor attempts to open a data file, such as a configuration file, an XML file, or a file containing serialized data, that it cannot find.

Regular expression match timeout values

You can set the default timeout value for a regular expression pattern matching operation on a per-application domain basis. The timeout is defined by specifying a `TimeSpan` value for the "REGEX_DEFAULT_MATCH_TIMEOUT" property to the `AppDomain.SetData` method. The time interval must be a valid `TimeSpan` object that is greater than zero and less than approximately 24 days. If these requirements are not met, the attempt to set the default timeout value throws an

[ArgumentOutOfRangeException](#), which in turn is wrapped in a [TypeInitializationException](#) exception.

The following example shows the [TypeInitializationException](#) that is thrown when the value assigned to the "REGEX_DEFAULT_MATCH_TIMEOUT" property is invalid. To eliminate the exception, set the "REGEX_DEFAULT_MATCH_TIMEOUT" property to a [TimeSpan](#) value that is greater than zero and less than approximately 24 days.

C#

```
using System;
using System.Text.RegularExpressions;

public class RegexEx1
{
    public static void Main()
    {
        AppDomain domain = AppDomain.CurrentDomain;
        // Set a timeout interval of -2 seconds.
        domain.SetData("REGEX_DEFAULT_MATCH_TIMEOUT",
TimeSpan.FromSeconds(-2));

        Regex rgx = new Regex("[aeiou]");
        Console.WriteLine("Regular expression pattern: {0}",
rgx.ToString());
        Console.WriteLine("Timeout interval for this regex: {0} seconds",
rgx.MatchTimeout.TotalSeconds);
    }
}
// The example displays the following output:
//     Unhandled Exception: System.TypeInitializationException:
//         The type initializer for 'System.Text.RegularExpressions.Regex'
threw an exception. --->
//         System.ArgumentOutOfRangeException: Specified argument was out of
the range of valid values.
//         Parameter name: AppDomain data 'REGEX_DEFAULT_MATCH_TIMEOUT'
contains an invalid value or
//         object for specifying a default matching timeout for
System.Text.RegularExpressions.Regex.
//         at System.Text.RegularExpressions.Regex.InitDefaultMatchTimeout()
//         at System.Text.RegularExpressions.Regex..cctor()
//         --- End of inner exception stack trace ---
//         at System.Text.RegularExpressions.Regex..ctor(String pattern)
//         at Example.Main()
```

Calendars and cultural data

If you attempt to instantiate a calendar but the runtime is unable to instantiate the [CultureInfo](#) object that corresponds to that calendar, it throws a

[TypeInitializationException](#) exception. This exception can be thrown by the following calendar class constructors:

- The parameterless constructor of the [JapaneseCalendar](#) class.
- The parameterless constructor of the [KoreanCalendar](#) class.
- The parameterless constructor of the [TaiwanCalendar](#) class.

Since cultural data for these cultures should be available on all systems, you should rarely, if ever, encounter this exception.

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

.NET is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

Numerics in .NET

Article • 09/01/2023

.NET provides a range of numeric integer and floating-point primitives, as well as:

- [System.Half](#), which represents a half-precision floating-point number.
- [System.Decimal](#), which represents a decimal floating-point number.
- [System.Numerics.BigInteger](#), which is an integral type with no theoretical upper or lower bound.
- [System.Numerics.Complex](#), which represents complex numbers.
- A set of SIMD-enabled types in the [System.Numerics](#) namespace.

Integer types

.NET supports both signed and unsigned 8-bit, 16-bit, 32-bit, 64-bit, and 128-bit integer types, which are listed in the following tables.

Signed integer types

| Type | Size (in bytes) | Minimum value | Maximum value |
|---|-----------------------|--|---|
| System.Int16 | 2 | -32,768 | 32,767 |
| System.Int32 | 4 | -2,147,483,648 | 2,147,483,647 |
| System.Int64 | 8 | -9,223,372,036,854,775,808 | 9,223,372,036,854,775,807 |
| System.Int128 | 16 | -170,141,183,460,469,231,731,687,303,715,884,105,728 | 170,141,183,460,469,231,731,687,303,715,884,105,727 |
| System.SByte | 1 | -128 | 127 |
| System.IntPtr (in 32-bit process) | 4 | -2,147,483,648 | 2,147,483,647 |
| System.IntPtr (in 64-bit process) | 8 | -9,223,372,036,854,775,808 | 9,223,372,036,854,775,807 |

Unsigned integer types

| Type | Size (in bytes) | Minimum value | Maximum value |
|--|-----------------|---------------|---|
| System.Byte | 1 | 0 | 255 |
| System.UInt16 | 2 | 0 | 65,535 |
| System.UInt32 | 4 | 0 | 4,294,967,295 |
| System.UInt64 | 8 | 0 | 18,446,744,073,709,551,615 |
| System.UInt128 | 16 | 0 | 340,282,366,920,938,463,463,374,607,431,768,211,455 |
| System.UIntPtr (in 32-bit process) | 4 | 0 | 4,294,967,295 |
| System.UIntPtr (in 64-bit process) | 8 | 0 | 18,446,744,073,709,551,615 |

Each integer type supports a set of standard arithmetic operators. The [System.Math](#) class provides methods for a broader set of mathematical functions.

You can also work with the individual bits in an integer value by using the [System.BitConverter](#) class.

ⓘ Note

The unsigned integer types are not CLS-compliant. For more information, see [Language independence and language-independent components](#).

BigInteger

The [System.Numerics.BigInteger](#) structure is an immutable type that represents an arbitrarily large integer whose value in theory has no upper or lower bounds. The methods of the [BigInteger](#) type closely parallel those of the other integral types.

Floating-point types

.NET includes the following floating-point types:

| Type | Size (in bytes) | Approximate range | Primitive? | Notes |
|--------------------------------|-----------------|-----------------------------|------------|----------------------|
| System.Half | 2 | ± 65504 | No | Introduced in .NET 5 |
| System.Single | 4 | $\pm 3.4 \times 10^{38}$ | Yes | |
| System.Double | 8 | $\pm 1.7 \times 10^{308}$ | Yes | |
| System.Decimal | 16 | $\pm 7.9228 \times 10^{28}$ | No | |

The [Half](#), [Single](#), and [Double](#) types support special values that represent not-a-number and infinity. For example, the [Double](#) type provides the following values: [Double.NaN](#), [Double.NegativeInfinity](#), and [Double.PositiveInfinity](#). You use the [Double.IsNaN](#), [Double.IsInfinity](#), [Double.IsPositiveInfinity](#), and [Double.IsNegativeInfinity](#) methods to test for these special values.

Each floating-point type supports a set of standard arithmetic operators. The [System.Math](#) class provides methods for a broader set of mathematical functions. .NET Core 2.0 and later includes the [System.MathF](#) class, which provides methods that accept arguments of the [Single](#) type.

You can also work with the individual bits in [Double](#), [Single](#), and [Half](#) values by using the [System.BitConverter](#) class. The [System.Decimal](#) structure has its own methods, [Decimal.GetBits](#) and [Decimal\(Int32\[\]\)](#), for working with a decimal value's individual bits, as well as its own set of methods for performing some additional mathematical operations.

The [Double](#), [Single](#), and [Half](#) types are intended to be used for values that, by their nature, are imprecise (for example, the distance between two stars) and for applications in which a high degree of precision and small rounding error is not required. Use the [System.Decimal](#) type for cases in which greater precision is required and rounding errors should be minimized.

ⓘ Note

The [Decimal](#) type doesn't eliminate the need for rounding. Rather, it minimizes errors due to rounding.

Complex

The [System.Numerics.Complex](#) structure represents a complex number, that is, a number with a real number part and an imaginary number part. It supports a standard set of arithmetic, comparison, equality, explicit and implicit conversion operators, as well as mathematical, algebraic, and trigonometric methods.

SIMD-enabled types

The [System.Numerics](#) namespace includes a set of .NET SIMD-enabled types. SIMD (Single Instruction Multiple Data) operations can be parallelized at the hardware level. That increases the throughput of the vectorized computations, which are common in mathematical, scientific, and graphics apps.

The .NET SIMD-enabled types include the following:

- The [Vector2](#), [Vector3](#), and [Vector4](#) types, which represent vectors with 2, 3, and 4 [Single](#) values.
- Two matrix types, [Matrix3x2](#), which represents a 3x2 matrix, and [Matrix4x4](#), which represents a 4x4 matrix.
- The [Plane](#) type, which represents a plane in three-dimensional space.
- The [Quaternion](#) type, which represents a vector that is used to encode three-dimensional physical rotations.
- The [Vector<T>](#) type, which represents a vector of a specified numeric type and provides a broad set of operators that benefit from SIMD support. The count of a [Vector<T>](#) instance is fixed, but its value [Vector<T>.Count](#) depends on the CPU of the machine, on which code is executed.

ⓘ Note

The [Vector<T>](#) type is included with .NET Core and .NET 5+, but not .NET Framework. If you're using .NET Framework, install the [System.Numerics.Vectors](#) NuGet package to get access to this type.

The SIMD-enabled types are implemented in such a way that they can be used with non-SIMD-enabled hardware or JIT compilers. To take advantage of SIMD instructions, your 64-bit apps must be run by the runtime that uses the RyuJIT compiler, which is included in .NET Core and in .NET Framework 4.6 and later versions. It adds SIMD support when targeting 64-bit processors.

For more information, see [Use SIMD-accelerated numeric types](#).

See also

- [Standard numeric format strings](#)
- [Floating-point numeric types in C#](#)

ⓘ Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



.NET feedback

The .NET documentation is open source. Provide feedback here.

ⓘ Open a documentation issue

ⓘ Provide product feedback

System.Boolean struct

Article • 01/08/2024

This article provides supplementary remarks to the reference documentation for this API.

A [Boolean](#) instance can have either of two values: `true` or `false`.

The [Boolean](#) structure provides methods that support the following tasks:

- Converting Boolean values to strings: [ToString](#)
- Parsing strings to convert them to Boolean values: [Parse](#) and [TryParse](#)
- Comparing values: [CompareTo](#) and [Equals](#)

This article explains these tasks and other usage details.

Format Boolean values

The string representation of a [Boolean](#) is either "True" for a `true` value or "False" for a `false` value. The string representation of a [Boolean](#) value is defined by the read-only [TrueString](#) and [FalseString](#) fields.

You use the [ToString](#) method to convert Boolean values to strings. The Boolean structure includes two [ToString](#) overloads: the parameterless [ToString\(\)](#) method and the [ToString\(IFormatProvider\)](#) method, which includes a parameter that controls formatting. However, because this parameter is ignored, the two overloads produce identical strings. The [ToString\(IFormatProvider\)](#) method does not support culture-sensitive formatting.

The following example illustrates formatting with the [ToString](#) method. Note that the C# and VB examples use the [composite formatting](#) feature, while the F# example uses [string interpolation](#). In both cases the [ToString](#) method is called implicitly.

C#

```
using System;

public class Example10
{
    public static void Main()
    {
        bool raining = false;
        bool busLate = true;
```

```

        Console.WriteLine("It is raining: {0}", raining);
        Console.WriteLine("The bus is late: {0}", busLate);
    }
}

// The example displays the following output:
//      It is raining: False
//      The bus is late: True

```

Because the **Boolean** structure can have only two values, it is easy to add custom formatting. For simple custom formatting in which other string literals are substituted for "True" and "False", you can use any conditional evaluation feature supported by your language, such as the [conditional operator](#) in C# or the [If operator](#) in Visual Basic. The following example uses this technique to format **Boolean** values as "Yes" and "No" rather than "True" and "False".

C#

```

using System;

public class Example11
{
    public static void Main()
    {
        bool raining = false;
        bool busLate = true;

        Console.WriteLine("It is raining: {0}",
                          raining ? "Yes" : "No");
        Console.WriteLine("The bus is late: {0}",
                          busLate ? "Yes" : "No");
    }
}

// The example displays the following output:
//      It is raining: No
//      The bus is late: Yes

```

For more complex custom formatting operations, including culture-sensitive formatting, you can call the [String.Format\(IFormatProvider, String, Object\[\]\)](#) method and provide an [ICustomFormatter](#) implementation. The following example implements the [ICustomFormatter](#) and [IFormatProvider](#) interfaces to provide culture-sensitive Boolean strings for the English (United States), French (France), and Russian (Russia) cultures.

C#

```

using System;
using System.Globalization;

public class Example4

```

```
{  
    public static void Main()  
    {  
        String[] cultureNames = { "", "en-US", "fr-FR", "ru-RU" };  
        foreach (var cultureName in cultureNames) {  
            bool value = true;  
            CultureInfo culture =  
CultureInfo.CreateSpecificCulture(cultureName);  
            BooleanFormatter formatter = new BooleanFormatter(culture);  
  
                string result = string.Format(formatter, "Value for '{0}': {1}",  
culture.Name, value);  
                Console.WriteLine(result);  
        }  
    }  
}  
  
public class BooleanFormatter : ICustomFormatter, IFormatProvider  
{  
    private CultureInfo culture;  
  
    public BooleanFormatter() : this(CultureInfo.CurrentCulture)  
    { }  
  
    public BooleanFormatter(CultureInfo culture)  
    {  
        this.culture = culture;  
    }  
  
    public Object GetFormat(Type formatType)  
    {  
        if (formatType == typeof(ICustomFormatter))  
            return this;  
        else  
            return null;  
    }  
  
    public string Format(string fmt, Object arg, IFormatProvider  
formatProvider)  
    {  
        // Exit if another format provider is used.  
        if (! formatProvider.Equals(this)) return null;  
  
        // Exit if the type to be formatted is not a Boolean  
        if (! (arg is Boolean)) return null;  
  
        bool value = (bool) arg;  
        switch (culture.Name) {  
            case "en-US":  
                return value.ToString();  
            case "fr-FR":  
                if (value)  
                    return "vrai";  
                else  
                    return "faux";  
        }  
    }  
}
```

```

        case "ru-RU":
            if (value)
                return "верно";
            else
                return "неверно";
        default:
            return value.ToString();
    }
}
// The example displays the following output:
//      Value for '' : True
//      Value for 'en-US' : True
//      Value for 'fr-FR' : vrai
//      Value for 'ru-RU' : верно

```

Optionally, you can use [resource files](#) to define culture-specific Boolean strings.

Convert to and from Boolean values

The [Boolean](#) structure implements the [IConvertible](#) interface. As a result, you can use the [Convert](#) class to perform conversions between a [Boolean](#) value and any other primitive type in .NET, or you can call the [Boolean](#) structure's explicit implementations. However, conversions between a [Boolean](#) and the following types are not supported, so the corresponding conversion methods throw an [InvalidOperationException](#) exception:

- Conversion between [Boolean](#) and [Char](#) (the [Convert.ToBoolean\(Char\)](#) and [Convert.ToChar\(Boolean\)](#) methods).
- Conversion between [Boolean](#) and [DateTime](#) (the [Convert.ToBoolean\(DateTime\)](#) and [Convert.ToDateTime\(Boolean\)](#) methods).

All conversions from integral or floating-point numbers to Boolean values convert non-zero values to `true` and zero values to `false`. The following example illustrates this by calling selected overloads of the [Convert.ToBoolean](#) class.

C#

```

using System;

public class Example2
{
    public static void Main()
    {
        Byte byteValue = 12;
        Console.WriteLine(Convert.ToBoolean(byteValue));
        Byte byteValue2 = 0;
        Console.WriteLine(Convert.ToBoolean(byteValue2));
    }
}

```

```

        int intValue = -16345;
        Console.WriteLine(Convert.ToBoolean(intValue));
        long longValue = 945;
        Console.WriteLine(Convert.ToBoolean(longValue));
        SByte sbyteValue = -12;
        Console.WriteLine(Convert.ToBoolean(sbyteValue));
        double dblValue = 0;
        Console.WriteLine(Convert.ToBoolean(dblValue));
        float sngValue = .0001f;
        Console.WriteLine(Convert.ToBoolean(sngValue));
    }
}

// The example displays the following output:
//      True
//      False
//      True
//      True
//      True
//      False
//      True

```

When converting from Boolean to numeric values, the conversion methods of the [Convert](#) class convert `true` to 1 and `false` to 0. However, Visual Basic conversion functions convert `true` to either 255 (for conversions to [Byte](#) values) or -1 (for all other numeric conversions). The following example converts `true` to numeric values by using a [Convert](#) method, and, in the case of the Visual Basic example, by using the Visual Basic language's own conversion operator.

C#

```

using System;

public class Example3
{
    public static void Main()
    {
        bool flag = true;

        byte byteValue;
        byteValue = Convert.ToByte(flag);
        Console.WriteLine("{0} -> {1}", flag, byteValue);

        sbyte sbyteValue;
        sbyteValue = Convert.ToSByte(flag);
        Console.WriteLine("{0} -> {1}", flag, sbyteValue);

        double dblValue;
        dblValue = Convert.ToDouble(flag);
        Console.WriteLine("{0} -> {1}", flag, dblValue);

        int intValue;

```

```

        intValue = Convert.ToInt32(flag);
        Console.WriteLine("{0} -> {1}", flag, intValue);
    }
}

// The example displays the following output:
//      True -> 1
//      True -> 1
//      True -> 1
//      True -> 1

```

For conversions from [Boolean](#) to string values, see the [Format Boolean values](#) section.
For conversions from strings to [Boolean](#) values, see the [Parse Boolean values](#) section.

Parse Boolean values

The [Boolean](#) structure includes two static parsing methods, [Parse](#) and [TryParse](#), that convert a string to a Boolean value. The string representation of a Boolean value is defined by the case-insensitive equivalents of the values of the [TrueString](#) and [FalseString](#) fields, which are "True" and "False", respectively. In other words, the only strings that parse successfully are "True", "False", "true", "false", or some mixed-case equivalent. You cannot successfully parse numeric strings such as "0" or "1". Leading or trailing white-space characters are not considered when performing the string comparison.

The following example uses the [Parse](#) and [TryParse](#) methods to parse a number of strings. Note that only the case-insensitive equivalents of "True" and "False" can be successfully parsed.

C#

```

using System;

public class Example7
{
    public static void Main()
    {
        string[] values = { null, String.Empty, "True", "False",
                           "true", "false", "    true    ",
                           "TrUe", "fAlSe", "fa lse", "0",
                           "1", "-1", "string" };
        // Parse strings using the Boolean.Parse method.
        foreach (var value in values) {
            try {
                bool flag = Boolean.Parse(value);
                Console.WriteLine("'{}' --> {}", value, flag);
            }
            catch (ArgumentException) {
                Console.WriteLine("Cannot parse a null string.");
            }
        }
    }
}

```

```

        }
        catch (FormatException) {
            Console.WriteLine("Cannot parse '{0}'.", value);
        }
    }
    Console.WriteLine();
    // Parse strings using the Boolean.TryParse method.
    foreach (var value in values) {
        bool flag = false;
        if (Boolean.TryParse(value, out flag))
            Console.WriteLine('{0}' --> {1}, value, flag);
        else
            Console.WriteLine("Unable to parse '{0}'", value);
    }
}
// The example displays the following output:
//     Cannot parse a null string.
//     Cannot parse ''.
//     'True' --> True
//     'False' --> False
//     'true' --> True
//     'false' --> False
//     '    true    ' --> True
//     'TrUe' --> True
//     'fAlSe' --> False
//     Cannot parse 'fa lse'.
//     Cannot parse '0'.
//     Cannot parse '1'.
//     Cannot parse '-1'.
//     Cannot parse 'string'.

//     Unable to parse ''
//     Unable to parse ''
//     'True' --> True
//     'False' --> False
//     'true' --> True
//     'false' --> False
//     '    true    ' --> True
//     'TrUe' --> True
//     'fAlSe' --> False
//     Cannot parse 'fa lse'.
//     Unable to parse '0'
//     Unable to parse '1'
//     Unable to parse '-1'
//     Unable to parse 'string'

```

If you're programming in Visual Basic, you can use the `CBool` function to convert the string representation of a number to a Boolean value. "0" is converted to `false`, and the string representation of any non-zero value is converted to `true`. If you're not programming in Visual Basic, you must convert your numeric string to a number before

converting it to a Boolean. The following example illustrates this by converting an array of integers to Boolean values.

```
C#  
  
using System;  
  
public class Example8  
{  
    public static void Main()  
    {  
        String[] values = { "09", "12.6", "0", "-13 " };  
        foreach (var value in values) {  
            bool success, result;  
            int number;  
            success = Int32.TryParse(value, out number);  
            if (success) {  
                // The method throws no exceptions.  
                result = Convert.ToBoolean(number);  
                Console.WriteLine("Converted '{0}' to {1}", value, result);  
            }  
            else {  
                Console.WriteLine("Unable to convert '{0}'", value);  
            }  
        }  
    }  
}  
  
// The example displays the following output:  
//     Converted '09' to True  
//     Unable to convert '12.6'  
//     Converted '0' to False  
//     Converted '-13 ' to True
```

Compare Boolean values

Because Boolean values are either `true` or `false`, there is little reason to explicitly call the [CompareTo](#) method, which indicates whether an instance is greater than, less than, or equal to a specified value. Typically, to compare two Boolean variables, you call the [Equals](#) method or use your language's equality operator.

However, when you want to compare a Boolean variable with the literal Boolean value `true` or `false`, it's not necessary to do an explicit comparison, because the result of evaluating a Boolean value is that Boolean value. For example, the following two expressions are equivalent, but the second is more compact. However, both techniques offer comparable performance.

```
C#
```

```
if (booleanValue == true) {
```

C#

```
if (booleanValue) {
```

Work with Booleans as binary values

A Boolean value occupies one byte of memory, as the following example shows. The C# example must be compiled with the `/unsafe` switch.

C#

```
using System;

public struct BoolStruct
{
    public bool flag1;
    public bool flag2;
    public bool flag3;
    public bool flag4;
    public bool flag5;
}

public class Example9
{
    public static void Main()
    {
        unsafe {
            BoolStruct b = new BoolStruct();
            bool* addr = (bool*) &b;
            Console.WriteLine("Size of BoolStruct: {0}", sizeof(BoolStruct));
            Console.WriteLine("Field offsets:");
            Console.WriteLine("    flag1: {0}", (bool*) &b.flag1 - addr);
            Console.WriteLine("    flag1: {0}", (bool*) &b.flag2 - addr);
            Console.WriteLine("    flag1: {0}", (bool*) &b.flag3 - addr);
            Console.WriteLine("    flag1: {0}", (bool*) &b.flag4 - addr);
            Console.WriteLine("    flag1: {0}", (bool*) &b.flag5 - addr);
        }
    }
}

// The example displays the following output:
//      Size of BoolStruct: 5
//      Field offsets:
//          flag1: 0
//          flag1: 1
//          flag1: 2
```

```
//           flag1: 3  
//           flag1: 4
```

The byte's low-order bit is used to represent its value. A value of 1 represents `true`; a value of 0 represents `false`.

Tip

You can use the `System.Collections.Specialized.BitVector32` structure to work with sets of Boolean values.

You can convert a Boolean value to its binary representation by calling the `BitConverter.GetBytes(Boolean)` method. The method returns a byte array with a single element. To restore a Boolean value from its binary representation, you can call the `BitConverter.ToBoolean(Byte[], Int32)` method.

The following example calls the `BitConverter.GetBytes` method to convert a Boolean value to its binary representation and displays the individual bits of the value, and then calls the `BitConverter.ToBoolean` method to restore the value from its binary representation.

C#

```
using System;  
  
public class Example1  
{  
    public static void Main()  
    {  
        bool[] flags = { true, false };  
        foreach (var flag in flags)  
        {  
            // Get binary representation of flag.  
            Byte value = BitConverter.GetBytes(flag)[0];  
            Console.WriteLine("Original value: {0}", flag);  
            Console.WriteLine("Binary value: {0} ({1})", value,  
                GetBinaryString(value));  
            // Restore the flag from its binary representation.  
            bool newFlag = BitConverter.ToBoolean(new Byte[] { value }, 0);  
            Console.WriteLine("Restored value: {0}\n", flag);  
        }  
    }  
  
    private static string GetBinaryString(Byte value)  
    {  
        string retVal = Convert.ToString(value, 2);  
        return new string('0', 8 - retVal.Length) + retVal;  
    }  
}
```

```
}

// The example displays the following output:
//      Original value: True
//      Binary value:   1 (00000001)
//      Restored value: True
//
//      Original value: False
//      Binary value:   0 (00000000)
//      Restored value: False
```

Perform operations with Boolean values

This section illustrates how Boolean values are used in apps. The first section discusses its use as a flag. The second illustrates its use for arithmetic operations.

Boolean values as flags

Boolean variables are most commonly used as flags, to signal the presence or absence of some condition. For example, in the [String.Compare\(String, String, Boolean\)](#) method, the final parameter, `ignoreCase`, is a flag that indicates whether the comparison of two strings is case-insensitive (`ignoreCase` is `true`) or case-sensitive (`ignoreCase` is `false`). The value of the flag can then be evaluated in a conditional statement.

The following example uses a simple console app to illustrate the use of Boolean variables as flags. The app accepts command-line parameters that enable output to be redirected to a specified file (the `/f` switch), and that enable output to be sent both to a specified file and to the console (the `/b` switch). The app defines a flag named `isRedirected` to indicate whether output is to be sent to a file, and a flag named `isBoth` to indicate that output should be sent to the console. The F# example uses a [recursive function](#) to parse the arguments.

C#

```
using System;
using System.IO;
using System.Threading;

public class Example5
{
    public static void Main()
    {
        // Initialize flag variables.
        bool isRedirected = false;
        bool isBoth = false;
        String fileName = "";
```

```
StreamWriter sw = null;

// Get any command line arguments.
String[] args = Environment.GetCommandLineArgs();
// Handle any arguments.
if (args.Length > 1) {
    for (int ctr = 1; ctr < args.Length; ctr++) {
        String arg = args[ctr];
        if (arg.StartsWith("/") || arg.StartsWith("-")) {
            switch (arg.Substring(1).ToLower())
            {
                case "f":
                    isRedirected = true;
                    if (args.Length < ctr + 2) {
                        ShowSyntax("The /f switch must be followed by a
filename.");
                        return;
                    }
                    fileName = args[ctr + 1];
                    ctr++;
                    break;
                case "b":
                    isBoth = true;
                    break;
                default:
                    ShowSyntax(String.Format("The {0} switch is not
supported",
args[ctr]));
                    return;
            }
        }
    }
}

// If isBoth is True, isRedirected must be True.
if (isBoth && !isRedirected) {
    ShowSyntax("The /f switch must be used if /b is used.");
    return;
}

// Handle output.
if (isRedirected) {
    sw = new StreamWriter(fileName);
    if (!isBoth)
        Console.SetOut(sw);
}
String msg = String.Format("Application began at {0}", DateTime.Now);
Console.WriteLine(msg);
if (isBoth) sw.WriteLine(msg);
Thread.Sleep(5000);
msg = String.Format("Application ended normally at {0}",
DateTime.Now);
Console.WriteLine(msg);
if (isBoth) sw.WriteLine(msg);
if (isRedirected) sw.Close();
```

```
}

private static void ShowSyntax(String errMsg)
{
    Console.WriteLine(errMsg);
    Console.WriteLine("\nSyntax: Example [[/f <filename> [/b]]\n");
}
}
```

Booleans and arithmetic operations

A Boolean value is sometimes used to indicate the presence of a condition that triggers a mathematical calculation. For example, a `hasShippingCharge` variable might serve as a flag to indicate whether to add shipping charges to an invoice amount.

Because an operation with a `false` value has no effect on the result of an operation, it's not necessary to convert the Boolean to an integral value to use in the mathematical operation. Instead, you can use conditional logic.

The following example computes an amount that consists of a subtotal, a shipping charge, and an optional service charge. The `hasServiceCharge` variable determines whether the service charge is applied. Instead of converting `hasServiceCharge` to a numeric value and multiplying it by the amount of the service charge, the example uses conditional logic to add the service charge amount if it is applicable.

C#

```
using System;

public class Example6
{
    public static void Main()
    {
        bool[] hasServiceCharges = { true, false };
        Decimal subtotal = 120.62m;
        Decimal shippingCharge = 2.50m;
        Decimal serviceCharge = 5.00m;

        foreach (var hasServiceCharge in hasServiceCharges) {
            Decimal total = subtotal + shippingCharge +
                (hasServiceCharge ? serviceCharge : 0);
            Console.WriteLine("hasServiceCharge = {1}: The total is {0:C2}.",
                total, hasServiceCharge);
        }
    }
}

// The example displays output like the following:
```

```
//      hasServiceCharge = True: The total is $128.12.  
//      hasServiceCharge = False: The total is $123.12.
```

Booleans and interop

While marshaling base data types to COM is generally straightforward, the `Boolean` data type is an exception. You can apply the `MarshalAsAttribute` attribute to marshal the `Boolean` type to any of the following representations:

[+] Expand table

| Enumeration type | Unmanaged format |
|--|--|
| <code>UnmanagedType.Bool</code> | A 4-byte integer value, where any nonzero value represents <code>true</code> and 0 represents <code>false</code> . This is the default format of a <code>Boolean</code> field in a structure and of a <code>Boolean</code> parameter in platform invoke calls. |
| <code>UnmanagedType.U1</code> | A 1-byte integer value, where the 1 represents <code>true</code> and 0 represents <code>false</code> . |
| <code>UnmanagedType.VariantBool</code> | A 2-byte integer value, where -1 represents <code>true</code> and 0 represents <code>false</code> . This is the default format of a <code>Boolean</code> parameter in COM interop calls. |

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

.NET is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

System.Byte struct

Article • 01/08/2024

This article provides supplementary remarks to the reference documentation for this API.

[Byte](#) is an immutable value type that represents unsigned integers with values that range from 0 (which is represented by the [Byte.MinValue](#) constant) to 255 (which is represented by the [Byte.MaxValue](#) constant). .NET also includes a signed 8-bit integer value type, [SByte](#), which represents values that range from -128 to 127.

Instantiate a Byte value

You can instantiate a [Byte](#) value in several ways:

- You can declare a [Byte](#) variable and assign it a literal integer value that is within the range of the [Byte](#) data type. The following example declares two [Byte](#) variables and assigns them values in this way.

```
C#
```

```
byte value1 = 64;
byte value2 = 255;
```

- You can assign a non-byte numeric value to a byte. This is a narrowing conversion, so it requires a cast operator in C# and F#, or a conversion method in Visual Basic if `Option Strict` is on. If the non-byte value is a [Single](#), [Double](#), or [Decimal](#) value that includes a fractional component, the handling of its fractional part depends on the compiler performing the conversion. The following example assigns several numeric values to [Byte](#) variables.

```
C#
```

```
int int1 = 128;
try
{
    byte value1 = (byte)int1;
    Console.WriteLine(value1);
}
catch (OverflowException)
{
    Console.WriteLine("{0} is out of range of a byte.", int1);
}
```

```

double dbl2 = 3.997;
try
{
    byte value2 = (byte)dbl2;
    Console.WriteLine(value2);
}
catch (OverflowException)
{
    Console.WriteLine("{0} is out of range of a byte.", dbl2);
}
// The example displays the following output:
//      128
//      3

```

- You can call a method of the [Convert](#) class to convert any supported type to a [Byte](#) value. This is possible because [Byte](#) supports the [IConvertible](#) interface. The following example illustrates the conversion of an array of [Int32](#) values to [Byte](#) values.

C#

```

int[] numbers = { Int32.MinValue, -1, 0, 121, 340, Int32.MaxValue };
byte result;
foreach (int number in numbers)
{
    try
    {
        result = Convert.ToByte(number);
        Console.WriteLine("Converted the {0} value {1} to the {2} value
{3}.",
                           number.GetType().Name, number,
                           result.GetType().Name, result);
    }
    catch (OverflowException)
    {
        Console.WriteLine("The {0} value {1} is outside the range of
the Byte type.",
                           number.GetType().Name, number);
    }
}
// The example displays the following output:
//      The Int32 value -2147483648 is outside the range of the Byte
//      type.
//      The Int32 value -1 is outside the range of the Byte type.
//      Converted the Int32 value 0 to the Byte value 0.
//      Converted the Int32 value 121 to the Byte value 121.
//      The Int32 value 340 is outside the range of the Byte type.
//      The Int32 value 2147483647 is outside the range of the Byte
//      type.

```

- You can call the [Parse](#) or [TryParse](#) method to convert the string representation of a [Byte](#) value to a [Byte](#). The string can contain either decimal or hexadecimal digits. The following example illustrates the parse operation by using both a decimal and a hexadecimal string.

C#

```
string string1 = "244";
try
{
    byte byte1 = Byte.Parse(string1);
    Console.WriteLine(byte1);
}
catch (OverflowException)
{
    Console.WriteLine("'{0}' is out of range of a byte.", string1);
}
catch (FormatException)
{
    Console.WriteLine("'{0}' is out of range of a byte.", string1);
}

string string2 = "F9";
try
{
    byte byte2 = Byte.Parse(string2,
        System.Globalization.NumberStyles.HexNumber);
    Console.WriteLine(byte2);
}
catch (OverflowException)
{
    Console.WriteLine("'{0}' is out of range of a byte.", string2);
}
catch (FormatException)
{
    Console.WriteLine("'{0}' is out of range of a byte.", string2);
}
// The example displays the following output:
//      244
//      249
```

Perform operations on Byte values

The [Byte](#) type supports standard mathematical operations such as addition, subtraction, division, multiplication, subtraction, negation, and unary negation. Like the other integral types, the [Byte](#) type also supports the bitwise [AND](#), [OR](#), [XOR](#), left shift, and right shift operators.

You can use the standard numeric operators to compare two [Byte](#) values, or you can call the [CompareTo](#) or [Equals](#) method.

You can also call the members of the [Math](#) class to perform a wide range of numeric operations, including getting the absolute value of a number, calculating the quotient and remainder from integral division, determining the maximum or minimum value of two integers, getting the sign of a number, and rounding a number.

Represent a Byte as a String

The [Byte](#) type provides full support for standard and custom numeric format strings. (For more information, see [Formatting Types](#), [Standard Numeric Format Strings](#), and [Custom Numeric Format Strings](#).) However, most commonly, byte values are represented as one-digit to three-digit values without any additional formatting, or as two-digit hexadecimal values.

To format a [Byte](#) value as an integral string with no leading zeros, you can call the parameterless [ToString\(\)](#) method. By using the "D" format specifier, you can also include a specified number of leading zeros in the string representation. By using the "X" format specifier, you can represent a [Byte](#) value as a hexadecimal string. The following example formats the elements in an array of [Byte](#) values in these three ways.

C#

```
byte[] numbers = { 0, 16, 104, 213 };
foreach (byte number in numbers)
{
    // Display value using default formatting.
    Console.Write("{0,-3} --> ", number.ToString());
    // Display value with 3 digits and leading zeros.
    Console.Write(number.ToString("D3") + " ");
    // Display value with hexadecimal.
    Console.Write(number.ToString("X2") + " ");
    // Display value with four hexadecimal digits.
    Console.WriteLine(number.ToString("X4"));
}
// The example displays the following output:
//      0    -->  000   00   0000
//     16    -->  016   10   0010
//    104    -->  104   68   0068
//   213    -->  213   D5   00D5
```

You can also format a [Byte](#) value as a binary, octal, decimal, or hexadecimal string by calling the [ToString\(Byte, Int32\)](#) method and supplying the base as the method's second

parameter. The following example calls this method to display the binary, octal, and hexadecimal representations of an array of byte values.

```
C#
```

```
byte[] numbers = { 0, 16, 104, 213 };
Console.WriteLine("{0} {1,8} {2,5} {3,5}",
                  "Value", "Binary", "Octal", "Hex");
foreach (byte number in numbers)
{
    Console.WriteLine("{0,5} {1,8} {2,5} {3,5}",
                      number, Convert.ToString(number, 2),
                      Convert.ToString(number, 8),
                      Convert.ToString(number, 16));
}
// The example displays the following output:
//      Value      Binary     Octal     Hex
//      0          0          0          0
//      16         10000      20         10
//      104        1101000    150        68
//      213        11010101   325        d5
```

Work with non-decimal Byte values

In addition to working with individual bytes as decimal values, you may want to perform bitwise operations with byte values, or work with byte arrays or with the binary or hexadecimal representations of byte values. For example, overloads of the [BitConverter.GetBytes](#) method can convert each of the primitive data types to a byte array, and the [BigInteger.ToByteArray](#) method converts a [BigInteger](#) value to a byte array.

[Byte](#) values are represented in 8 bits by their magnitude only, without a sign bit. This is important to keep in mind when you perform bitwise operations on [Byte](#) values or when you work with individual bits. To perform a numeric, Boolean, or comparison operation on any two non-decimal values, both values must use the same representation.

When an operation is performed on two [Byte](#) values, the values share the same representation, so the result is accurate. This is illustrated in the following example, which masks the lowest-order bit of a [Byte](#) value to ensure that it is even.

```
C#
```

```
using System;
using System.Globalization;

public class Example
{
```

```

public static void Main()
{
    string[] values = { Convert.ToString(12, 16),
                        Convert.ToString(123, 16),
                        Convert.ToString(245, 16) };

    byte mask = 0xFE;
    foreach (string value in values) {
        Byte byteValue = Byte.Parse(value, NumberStyles.AllowHexSpecifier);
        Console.WriteLine("{0} And {1} = {2}", byteValue, mask,
                          byteValue & mask);
    }
}
// The example displays the following output:
//      12 And 254 = 12
//      123 And 254 = 122
//      245 And 254 = 244

```

On the other hand, when you work with both unsigned and signed bits, bitwise operations are complicated by the fact that the `SByte` values use sign-and-magnitude representation for positive values, and two's complement representation for negative values. In order to perform a meaningful bitwise operation, the values must be converted to two equivalent representations, and information about the sign bit must be preserved. The following example does this to mask out bits 2 and 4 of an array of 8-bit signed and unsigned values.

C#

```

using System;
using System.Collections.Generic;
using System.Globalization;

public struct ByteString
{
    public string Value;
    public int Sign;
}

public class Example1
{
    public static void Main()
    {
        ByteString[] values = CreateArray(-15, 123, 245);

        byte mask = 0x14;           // Mask all bits but 2 and 4.

        foreach (ByteString strValue in values)
        {
            byte byteValue = Byte.Parse(strValue.Value,
NumberStyles.AllowHexSpecifier);

```

```

        Console.WriteLine("{0} ({1}) And {2} ({3}) = {4} ({5})",
            strValue.Sign * byteValue,
            Convert.ToString(byteValue, 2),
            mask, Convert.ToString(mask, 2),
            (strValue.Sign & Math.Sign(mask)) * (byteValue
& mask),
            Convert.ToString(byteValue & mask, 2));
    }
}

private static ByteString[] CreateArray(params int[] values)
{
    List<ByteString> byteStrings = new List<ByteString>();

    foreach (object value in values)
    {
        ByteString temp = new ByteString();
        int sign = Math.Sign((int)value);
        temp.Sign = sign;

        // Change two's complement to magnitude-only representation.
        temp.Value = Convert.ToString(((int)value) * sign, 16);

        byteStrings.Add(temp);
    }
    return byteStrings.ToArray();
}
// The example displays the following output:
//      -15 (1111) And 20 (10100) = 4 (100)
//      123 (1111011) And 20 (10100) = 16 (10000)
//      245 (11110101) And 20 (10100) = 20 (10100)

```

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



.NET feedback

.NET is an open source project.
Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

System.Decimal struct

Article • 01/08/2024

This article provides supplementary remarks to the reference documentation for this API.

The [Decimal](#) value type represents decimal numbers ranging from positive 79,228,162,514,264,337,593,543,950,335 to negative 79,228,162,514,264,337,593,543,950,335. The default value of a [Decimal](#) is 0. The [Decimal](#) value type is appropriate for financial calculations that require large numbers of significant integral and fractional digits and no round-off errors. The [Decimal](#) type does not eliminate the need for rounding. Rather, it minimizes errors due to rounding. For example, the following code produces a result of 0.99999999999999999999999 instead of 1.

C#

```
decimal dividend = Decimal.One;
decimal divisor = 3;
// The following displays 0.9999999999999999999999 to the console
Console.WriteLine(dividend/divisor * divisor);
```

When the result of the division and multiplication is passed to the [Round](#) method, the result suffers no loss of precision, as the following code shows.

C#

```
decimal dividend = Decimal.One;
decimal divisor = 3;
// The following displays 1.00 to the console
Console.WriteLine(Math.Round(dividend/divisor * divisor, 2));
```

A decimal number is a floating-point value that consists of a sign, a numeric value where each digit in the value ranges from 0 to 9, and a scaling factor that indicates the position of a floating decimal point that separates the integral and fractional parts of the numeric value.

The binary representation of a [Decimal](#) value is 128-bits consisting of a 96-bit integer number, and a 32-bit set of flags representing things such as the sign and scaling factor used to specify what portion of it is a decimal fraction. Therefore, the binary representation of a [Decimal](#) value has the form, $((-2^{96} \text{ to } 2^{96}) / 10^{(0 \text{ to } 28)})$, where $-(2^{96}-1)$ is equal to [MinValue](#), and $2^{96}-1$ is equal to [MaxValue](#). For more information about the

binary representation of [Decimal](#) values and an example, see the [Decimal\(Int32\[\]\)](#) constructor and the [GetBits](#) method.

The scaling factor also preserves any trailing zeros in a [Decimal](#) number. Trailing zeros do not affect the value of a [Decimal](#) number in arithmetic or comparison operations. However, trailing zeros might be revealed by the [ToString](#) method if an appropriate format string is applied.

Conversion considerations

This type provides methods that convert [Decimal](#) values to and from [SByte](#), [Int16](#), [Int32](#), [Int64](#), [Byte](#), [UInt16](#), [UInt32](#), and [UInt64](#) values. Conversions from these integral types to [Decimal](#) are widening conversions that never lose information or throw exceptions.

Conversions from [Decimal](#) to any of the integral types are narrowing conversions that round the [Decimal](#) value to the nearest integer value toward zero. Some languages, such as C#, also support the conversion of [Decimal](#) values to [Char](#) values. If the result of these conversions cannot be represented in the destination type, an [OverflowException](#) exception is thrown.

The [Decimal](#) type also provides methods that convert [Decimal](#) values to and from [Single](#) and [Double](#) values. Conversions from [Decimal](#) to [Single](#) or [Double](#) are narrowing conversions that might lose precision but not information about the magnitude of the converted value. The conversion does not throw an exception.

Conversions from [Single](#) or [Double](#) to [Decimal](#) throw an [OverflowException](#) exception if the result of the conversion cannot be represented as a [Decimal](#).

Perform operations on [Decimal](#) values

The [Decimal](#) type supports standard mathematical operations such as addition, subtraction, division, multiplication, and unary negation. You can also work directly with the binary representation of a [Decimal](#) value by calling the [GetBits](#) method.

To compare two [Decimal](#) values, you can use the standard numeric comparison operators, or you can call the [CompareTo](#) or [Equals](#) method.

You can also call the members of the [Math](#) class to perform a wide range of numeric operations, including getting the absolute value of a number, determining the maximum or minimum value of two [Decimal](#) values, getting the sign of a number, and rounding a number.

Examples

The following code example demonstrates the use of [Decimal](#).

C#

```
/// <summary>
/// Keeping my fortune in Decimals to avoid the round-off errors.
/// </summary>
class PiggyBank {
    protected decimal MyFortune;

    public void AddPenny() {
        MyFortune = Decimal.Add(MyFortune, .01m);
    }

    public decimal Capacity {
        get {
            return Decimal.MaxValue;
        }
    }

    public decimal Dollars {
        get {
            return Decimal.Floor(MyFortune);
        }
    }

    public decimal Cents {
        get {
            return Decimal.Subtract(MyFortune, Decimal.Floor(MyFortune));
        }
    }

    public override string ToString() {
        return MyFortune.ToString("C")+" in piggy bank";
    }
}
```

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For

.NET

 .NET feedback

.NET is an open source project. Select a link to provide feedback:

 Open a documentation issue

 Provide product feedback

more information, see [our contributor guide](#).

System.Double struct

Article • 01/08/2024

This article provides supplementary remarks to the reference documentation for this API.

The [Double](#) value type represents a double-precision 64-bit number with values ranging from negative 1.79769313486232e308 to positive 1.79769313486232e308, as well as positive or negative zero, [PositiveInfinity](#), [NegativeInfinity](#), and not a number ([NaN](#)). It is intended to represent values that are extremely large (such as distances between planets or galaxies) or extremely small (such as the molecular mass of a substance in kilograms) and that often are imprecise (such as the distance from earth to another solar system). The [Double](#) type complies with the IEC 60559:1989 (IEEE 754) standard for binary floating-point arithmetic.

Floating-point representation and precision

The [Double](#) data type stores double-precision floating-point values in a 64-bit binary format, as shown in the following table:

[] Expand table

| Part | Bits |
|-----------------------------------|-------|
| Significand or mantissa | 0-51 |
| Exponent | 52-62 |
| Sign (0 = Positive, 1 = Negative) | 63 |

Just as decimal fractions are unable to precisely represent some fractional values (such as 1/3 or [Math.PI](#)), binary fractions are unable to represent some fractional values. For example, 1/10, which is represented precisely by .1 as a decimal fraction, is represented by .001100110011 as a binary fraction, with the pattern "0011" repeating to infinity. In this case, the floating-point value provides an imprecise representation of the number that it represents. Performing additional mathematical operations on the original floating-point value often tends to increase its lack of precision. For example, if we compare the result of multiplying .1 by 10 and adding .1 to .1 nine times, we see that addition, because it has involved eight more operations, has produced the less precise result. Note that this disparity is apparent only if we display the two [Double](#) values by

using the "R" standard numeric format string, which if necessary displays all 17 digits of precision supported by the `Double` type.

C#

```
using System;

public class Example13
{
    public static void Main()
    {
        Double value = .1;
        Double result1 = value * 10;
        Double result2 = 0;
        for (int ctr = 1; ctr <= 10; ctr++)
            result2 += value;

        Console.WriteLine(".1 * 10: {0:R}", result1);
        Console.WriteLine(".1 Added 10 times: {0:R}", result2);
    }
}

// The example displays the following output:
//      .1 * 10:      1
//      .1 Added 10 times: 0.9999999999999998
```

Because some numbers cannot be represented exactly as fractional binary values, floating-point numbers can only approximate real numbers.

All floating-point numbers also have a limited number of significant digits, which also determines how accurately a floating-point value approximates a real number. A `Double` value has up to 15 decimal digits of precision, although a maximum of 17 digits is maintained internally. This means that some floating-point operations may lack the precision to change a floating point value. The following example provides an illustration. It defines a very large floating-point value, and then adds the product of `Double.Epsilon` and one quadrillion to it. The product, however, is too small to modify the original floating-point value. Its least significant digit is thousandths, whereas the most significant digit in the product is 10^{-309} .

C#

```
using System;

public class Example14
{
    public static void Main()
    {
        Double value = 123456789012.34567;
        Double additional = Double.Epsilon * 1e15;
        Console.WriteLine("{0} + {1} = {2}", value, additional,
```

```

        value + additional);
    }
}

// The example displays the following output:
//      123456789012.346 + 4.94065645841247E-309 = 123456789012.346

```

The limited precision of a floating-point number has several consequences:

- Two floating-point numbers that appear equal for a particular precision might not compare equal because their least significant digits are different. In the following example, a series of numbers are added together, and their total is compared with their expected total. Although the two values appear to be the same, a call to the `Equals` method indicates that they are not.

```

C#

using System;

public class Example10
{
    public static void Main()
    {
        Double[] values = { 10.0, 2.88, 2.88, 2.88, 9.0 };
        Double result = 27.64;
        Double total = 0;
        foreach (var value in values)
            total += value;

        if (total.Equals(result))
            Console.WriteLine("The sum of the values equals the
total.");
        else
            Console.WriteLine("The sum of the values ({0}) does not
equal the total ({1}).",
                            total, result);
    }
}

// The example displays the following output:
//      The sum of the values (36.64) does not equal the total (36.64).
//
// If the index items in the Console.WriteLine statement are changed to
// {0:R},
// the example displays the following output:
//      The sum of the values (27.63999999999997) does not equal the
total (27.64).

```

If you change the format items in the `Console.WriteLine(String, Object, Object)` statement from `{0}` and `{1}` to `{0:R}` and `{1:R}` to display all significant digits of the two `Double` values, it is clear that the two values are unequal because of a loss

of precision during the addition operations. In this case, the issue can be resolved by calling the `Math.Round(Double, Int32)` method to round the `Double` values to the desired precision before performing the comparison.

- A mathematical or comparison operation that uses a floating-point number might not yield the same result if a decimal number is used, because the binary floating-point number might not equal the decimal number. A previous example illustrated this by displaying the result of multiplying .1 by 10 and adding .1 times.

When accuracy in numeric operations with fractional values is important, you can use the `Decimal` rather than the `Double` type. When accuracy in numeric operations with integral values beyond the range of the `Int64` or `UInt64` types is important, use the `BigInteger` type.

- A value might not round-trip if a floating-point number is involved. A value is said to round-trip if an operation converts an original floating-point number to another form, an inverse operation transforms the converted form back to a floating-point number, and the final floating-point number is not equal to the original floating-point number. The round trip might fail because one or more least significant digits are lost or changed in a conversion. In the following example, three `Double` values are converted to strings and saved in a file. As the output shows, however, even though the values appear to be identical, the restored values are not equal to the original values.

```
C#  
  
using System;  
using System.IO;  
  
public class Example11  
{  
    public static void Main()  
    {  
        StreamWriter sw = new StreamWriter(@".\Doubles.dat");  
        Double[] values = { 2.2 / 1.01, 1.0 / 3, Math.PI };  
        for (int ctr = 0; ctr < values.Length; ctr++)  
        {  
            sw.Write(values[ctr].ToString());  
            if (ctr != values.Length - 1)  
                sw.Write("|");  
        }  
        sw.Close();  
  
        Double[] restoredValues = new Double[values.Length];  
        StreamReader sr = new StreamReader(@".\Doubles.dat");  
        string temp = sr.ReadToEnd();  
        string[] tempStrings = temp.Split('|');  
        for (int ctr = 0; ctr < tempStrings.Length; ctr++)
```

```

        restoredValues[ctr] = Double.Parse(tempStrings[ctr]);

        for (int ctr = 0; ctr < values.Length; ctr++)
            Console.WriteLine("{0} {2} {1}", values[ctr],
                               restoredValues[ctr],
                               values[ctr].Equals(restoredValues[ctr]) ?
                               "=" : "<>");
    }
}

// The example displays the following output:
//      2.17821782178218 <> 2.17821782178218
//      0.3333333333333333 <> 0.3333333333333333
//      3.14159265358979 <> 3.14159265358979

```

In this case, the values can be successfully round-tripped by using the "G17" standard numeric format string to preserve the full precision of `Double` values, as the following example shows.

C#

```

using System;
using System.IO;

public class Example12
{
    public static void Main()
    {
        StreamWriter sw = new StreamWriter(@".\Doubles.dat");
        Double[] values = { 2.2 / 1.01, 1.0 / 3, Math.PI };
        for (int ctr = 0; ctr < values.Length; ctr++)
            sw.Write("{0:G17}{1}", values[ctr], ctr < values.Length - 1
? "|" : "");

        sw.Close();

        Double[] restoredValues = new Double[values.Length];
        StreamReader sr = new StreamReader(@".\Doubles.dat");
        string temp = sr.ReadToEnd();
        string[] tempStrings = temp.Split('|');
        for (int ctr = 0; ctr < tempStrings.Length; ctr++)
            restoredValues[ctr] = Double.Parse(tempStrings[ctr]);

        for (int ctr = 0; ctr < values.Length; ctr++)
            Console.WriteLine("{0} {2} {1}", values[ctr],
                               restoredValues[ctr],
                               values[ctr].Equals(restoredValues[ctr]) ?
                               "=" : "<>");
    }
}

// The example displays the following output:
//      2.17821782178218 = 2.17821782178218

```

```
//      0.3333333333333333 = 0.3333333333333333  
//      3.14159265358979 = 3.14159265358979
```

ⓘ Important

When used with a **Double** value, the "R" format specifier in some cases fails to successfully round-trip the original value. To ensure that **Double** values successfully round-trip, use the "G17" format specifier.

- Single values have less precision than Double values. A **Single** value that is converted to a seemingly equivalent **Double** often does not equal the **Double** value because of differences in precision. In the following example, the result of identical division operations is assigned to a **Double** and a **Single** value. After the **Single** value is cast to a **Double**, a comparison of the two values shows that they are unequal.

C#

```
using System;  
  
public class Example9  
{  
    public static void Main()  
    {  
        Double value1 = 1 / 3.0;  
        Single sValue2 = 1 / 3.0f;  
        Double value2 = (Double)sValue2;  
        Console.WriteLine("{0:R} = {1:R}: {2}", value1, value2,  
                           value1.Equals(value2));  
    }  
}  
// The example displays the following output:  
//      0.3333333333333331 = 0.333333432674408: False
```

To avoid this problem, use either the **Double** in place of the **Single** data type, or use the **Round** method so that both values have the same precision.

In addition, the result of arithmetic and assignment operations with **Double** values may differ slightly by platform because of the loss of precision of the **Double** type. For example, the result of assigning a literal **Double** value may differ in the 32-bit and 64-bit versions of .NET. The following example illustrates this difference when the literal value -4.42330604244772E-305 and a variable whose value is -4.42330604244772E-305 are assigned to a **Double** variable. Note that the result of the **Parse(String)** method in this case does not suffer from a loss of precision.

C#

```
double value = -4.42330604244772E-305;

double fromLiteral = -4.42330604244772E-305;
double fromVariable = value;
double fromParse = Double.Parse("-4.42330604244772E-305");

Console.WriteLine("Double value from literal: {0,29:R}", fromLiteral);
Console.WriteLine("Double value from variable: {0,28:R}", fromVariable);
Console.WriteLine("Double value from Parse method: {0,24:R}", fromParse);
// On 32-bit versions of the .NET Framework, the output is:
//    Double value from literal:      -4.42330604244772E-305
//    Double value from variable:     -4.42330604244772E-305
//    Double value from Parse method: -4.42330604244772E-305
//
// On other versions of the .NET Framework, the output is:
//    Double value from literal:      -4.4233060424477198E-305
//    Double value from variable:     -4.4233060424477198E-305
//    Double value from Parse method: -4.42330604244772E-305
```

Test for equality

To be considered equal, two [Double](#) values must represent identical values. However, because of differences in precision between values, or because of a loss of precision by one or both values, floating-point values that are expected to be identical often turn out to be unequal because of differences in their least significant digits. As a result, calls to the [Equals](#) method to determine whether two values are equal, or calls to the [CompareTo](#) method to determine the relationship between two [Double](#) values, often yield unexpected results. This is evident in the following example, where two apparently equal [Double](#) values turn out to be unequal because the first has 15 digits of precision, while the second has 17.

C#

```
using System;

public class Example
{
    public static void Main()
    {
        double value1 = .3333333333333333;
        double value2 = 1.0/3;
        Console.WriteLine("{0:R} = {1:R}: {2}", value1, value2,
value1.Equals(value2));
    }
}
```

```
// The example displays the following output:  
//      0.33333333333333 = 0.3333333333333331: False
```

Calculated values that follow different code paths and that are manipulated in different ways often prove to be unequal. In the following example, one `Double` value is squared, and then the square root is calculated to restore the original value. A second `Double` is multiplied by 3.51 and squared before the square root of the result is divided by 3.51 to restore the original value. Although the two values appear to be identical, a call to the `Equals(Double)` method indicates that they are not equal. Using the "R" standard format string to return a result string that displays all the significant digits of each `Double` value shows that the second value is .000000000001 less than the first.

C#

```
using System;  
  
public class Example1  
{  
    public static void Main()  
    {  
        double value1 = 100.10142;  
        value1 = Math.Sqrt(Math.Pow(value1, 2));  
        double value2 = Math.Pow(value1 * 3.51, 2);  
        value2 = Math.Sqrt(value2) / 3.51;  
        Console.WriteLine("{0} = {1}: {2}\n",
                           value1, value2, value1.Equals(value2));
        Console.WriteLine("{0:R} = {1:R}", value1, value2);
    }
}  
// The example displays the following output:  
//      100.10142 = 100.10142: False  
//  
//      100.10142 = 100.10141999999999
```

In cases where a loss of precision is likely to affect the result of a comparison, you can adopt any of the following alternatives to calling the `Equals` or `CompareTo` method:

- Call the `Math.Round` method to ensure that both values have the same precision. The following example modifies a previous example to use this approach so that two fractional values are equivalent.

C#

```
using System;  
  
public class Example2  
{  
    public static void Main()
```

```

    {
        double value1 = .33333333333333;
        double value2 = 1.0 / 3;
        int precision = 7;
        value1 = Math.Round(value1, precision);
        value2 = Math.Round(value2, precision);
        Console.WriteLine("{0:R} = {1:R}: {2}", value1, value2,
value1.Equals(value2));
    }
}
// The example displays the following output:
//      0.3333333 = 0.3333333: True

```

The problem of precision still applies to rounding of midpoint values. For more information, see the [Math.Round\(Double, Int32, MidpointRounding\)](#) method.

- Test for approximate equality rather than equality. This requires that you define either an absolute amount by which the two values can differ but still be equal, or that you define a relative amount by which the smaller value can diverge from the larger value.

Warning

Double.Epsilon is sometimes used as an absolute measure of the distance between two **Double** values when testing for equality. However, **Double.Epsilon** measures the smallest possible value that can be added to, or subtracted from, a **Double** whose value is zero. For most positive and negative **Double** values, the value of **Double.Epsilon** is too small to be detected. Therefore, except for values that are zero, we do not recommend its use in tests for equality.

The following example uses the latter approach to define an `IsApproximatelyEqual` method that tests the relative difference between two values. It also contrasts the result of calls to the `IsApproximatelyEqual` method and the `Equals(Double)` method.

C#

```

using System;

public class Example3
{
    public static void Main()
    {
        double one1 = .1 * 10;
        double one2 = 0;

```

```

        for (int ctr = 1; ctr <= 10; ctr++)
            one2 += .1;

            Console.WriteLine("{0:R} = {1:R}: {2}", one1, one2,
one1.Equals(one2));
            Console.WriteLine("{0:R} is approximately equal to {1:R}: {2}",
one1, one2,
IsApproximatelyEqual(one1, one2,
.00000001));
    }

    static bool IsApproximatelyEqual(double value1, double value2,
double epsilon)
{
    // If they are equal anyway, just return True.
    if (value1.Equals(value2))
        return true;

    // Handle NaN, Infinity.
    if (Double.IsInfinity(value1) | Double.IsNaN(value1))
        return value1.Equals(value2);
    else if (Double.IsInfinity(value2) | Double.IsNaN(value2))
        return value1.Equals(value2);

    // Handle zero to avoid division by zero
    double divisor = Math.Max(value1, value2);
    if (divisor.Equals(0))
        divisor = Math.Min(value1, value2);

    return Math.Abs((value1 - value2) / divisor) <= epsilon;
}
// The example displays the following output:
//      1 = 0.9999999999999989: False
//      1 is approximately equal to 0.9999999999999989: True

```

Floating-point values and exceptions

Unlike operations with integral types, which throw exceptions in cases of overflow or illegal operations such as division by zero, operations with floating-point values do not throw exceptions. Instead, in exceptional situations, the result of a floating-point operation is zero, positive infinity, negative infinity, or not a number (NaN):

- If the result of a floating-point operation is too small for the destination format, the result is zero. This can occur when two very small numbers are multiplied, as the following example shows.

```

using System;

public class Example6
{
    public static void Main()
    {
        Double value1 = 1.1632875981534209e-225;
        Double value2 = 9.1642346778e-175;
        Double result = value1 * value2;
        Console.WriteLine("{0} * {1} = {2}", value1, value2, result);
        Console.WriteLine("{0} = 0: {1}", result, result.Equals(0.0));
    }
}
// The example displays the following output:
//      1.16328759815342E-225 * 9.1642346778E-175 = 0
//      0 = 0: True

```

- If the magnitude of the result of a floating-point operation exceeds the range of the destination format, the result of the operation is [PositiveInfinity](#) or [NegativeInfinity](#), as appropriate for the sign of the result. The result of an operation that overflows [Double.MaxValue](#) is [PositiveInfinity](#), and the result of an operation that overflows [Double.MinValue](#) is [NegativeInfinity](#), as the following example shows.

C#

```

using System;

public class Example7
{
    public static void Main()
    {
        Double value1 = 4.565e153;
        Double value2 = 6.9375e172;
        Double result = value1 * value2;
        Console.WriteLine("PositiveInfinity: {0}",
                          Double.IsPositiveInfinity(result));
        Console.WriteLine("NegativeInfinity: {0}\n",
                          Double.IsNegativeInfinity(result));

        value1 = -value1;
        result = value1 * value2;
        Console.WriteLine("PositiveInfinity: {0}",
                          Double.IsPositiveInfinity(result));
        Console.WriteLine("NegativeInfinity: {0}",
                          Double.IsNegativeInfinity(result));
    }
}

// The example displays the following output:
//      PositiveInfinity: True

```

```
//      NegativeInfinity: False  
//  
//      PositiveInfinity: False  
//      NegativeInfinity: True
```

`PositiveInfinity` also results from a division by zero with a positive dividend, and `NegativeInfinity` results from a division by zero with a negative dividend.

- If a floating-point operation is invalid, the result of the operation is `NaN`. For example, `NaN` results from the following operations:
 - Division by zero with a dividend of zero. Note that other cases of division by zero result in either `PositiveInfinity` or `NegativeInfinity`.
 - Any floating-point operation with an invalid input. For example, calling the `Math.Sqrt` method with a negative value returns `NaN`, as does calling the `Math.Acos` method with a value that is greater than one or less than negative one.
 - Any operation with an argument whose value is `Double.NaN`.

Type conversions

The [Double](#) structure does not define any explicit or implicit conversion operators; instead, conversions are implemented by the compiler.

The conversion of the value of any primitive numeric type to a [Double](#) is a widening conversion and therefore does not require an explicit cast operator or call to a conversion method unless a compiler explicitly requires it. For example, the C# compiler requires a casting operator for conversions from [Decimal](#) to [Double](#), while the Visual Basic compiler does not. The following example converts the minimum or maximum value of other primitive numeric types to a [Double](#).

C#

```

UInt16.MinValue,
                           UInt16.MaxValue, UInt32.MinValue,
UInt32.MaxValue,
                           UInt64.MinValue, UInt64.MaxValue };

        double dblValue;
        foreach (var value in values)
        {
            if (value.GetType() == typeof(Decimal))
                dblValue = (Double)value;
            else
                dblValue = value;
            Console.WriteLine("{0} ({1}) --> {2:R} ({3})",
                            value, value.GetType().Name,
                            dblValue, dblValue.GetType().Name);
        }
    }
}

// The example displays the following output:
//     0 (Byte) --> 0 (Double)
//     255 (Byte) --> 255 (Double)
//     -79228162514264337593543950335 (Decimal) --> -7.9228162514264338E+28
// (Double)
//     79228162514264337593543950335 (Decimal) --> 7.9228162514264338E+28
// (Double)
//     -32768 (Int16) --> -32768 (Double)
//     32767 (Int16) --> 32767 (Double)
//     -2147483648 (Int32) --> -2147483648 (Double)
//     2147483647 (Int32) --> 2147483647 (Double)
//     -9223372036854775808 (Int64) --> -9.2233720368547758E+18 (Double)
//     9223372036854775807 (Int64) --> 9.2233720368547758E+18 (Double)
//     -128 (SByte) --> -128 (Double)
//     127 (SByte) --> 127 (Double)
//     -3.402823E+38 (Single) --> -3.4028234663852886E+38 (Double)
//     3.402823E+38 (Single) --> 3.4028234663852886E+38 (Double)
//     0 (UInt16) --> 0 (Double)
//     65535 (UInt16) --> 65535 (Double)
//     0 (UInt32) --> 0 (Double)
//     4294967295 (UInt32) --> 4294967295 (Double)
//     0 (UInt64) --> 0 (Double)
//     18446744073709551615 (UInt64) --> 1.8446744073709552E+19 (Double)

```

In addition, the [Single](#) values [Single.NaN](#), [Single.PositiveInfinity](#), and [Single.NegativeInfinity](#) convert to [Double.NaN](#), [Double.PositiveInfinity](#), and [Double.NegativeInfinity](#), respectively.

Note that the conversion of the value of some numeric types to a [Double](#) value can involve a loss of precision. As the example illustrates, a loss of precision is possible when converting [Decimal](#), [Int64](#), and [UInt64](#) values to [Double](#) values.

The conversion of a [Double](#) value to a value of any other primitive numeric data type is a narrowing conversion and requires a cast operator (in C#), a conversion method (in

Visual Basic), or a call to a [Convert](#) method. Values that are outside the range of the target data type, which are defined by the target type's `MinValue` and `MaxValue` properties, behave as shown in the following table.

[+] Expand table

| Target type | Result |
|-------------------|---|
| Any integral type | An OverflowException exception if the conversion occurs in a checked context. If the conversion occurs in an unchecked context (the default in C#), the conversion operation succeeds but the value overflows. |
| Decimal | An OverflowException exception. |
| Single | Single.NegativeInfinity for negative values. Single.PositiveInfinity for positive values. |

In addition, `Double.NaN`, `Double.PositiveInfinity`, and `Double.NegativeInfinity` throw an [OverflowException](#) for conversions to integers in a checked context, but these values overflow when converted to integers in an unchecked context. For conversions to `Decimal`, they always throw an [OverflowException](#). For conversions to `Single`, they convert to `Single.NaN`, `Single.PositiveInfinity`, and `Single.NegativeInfinity`, respectively.

A loss of precision may result from converting a `Double` value to another numeric type. In the case of converting to any of the integral types, as the output from the example shows, the fractional component is lost when the `Double` value is either rounded (as in Visual Basic) or truncated (as in C#). For conversions to `Decimal` and `Single` values, the `Double` value may not have a precise representation in the target data type.

The following example converts a number of `Double` values to several other numeric types. The conversions occur in a checked context in Visual Basic (the default), in C# (because of the `checked` keyword), and in F# (because of the `Checked` module). The output from the example shows the result for conversions in both a checked and unchecked context. You can perform conversions in an unchecked context in Visual Basic by compiling with the `/removeintchecks+` compiler switch, in C# by commenting out the `checked` statement, and in F# by commenting out the `open Checked` statement.

C#

```
using System;

public class Example5
{
    public static void Main()
```

```
{  
    Double[] values = { Double.MinValue, -67890.1234, -12345.6789,  
                        12345.6789, 67890.1234, Double.MaxValue,  
                        Double.NaN, Double.PositiveInfinity,  
                        Double.NegativeInfinity };  
    checked  
    {  
        foreach (var value in values)  
        {  
            try  
            {  
                Int64 lValue = (long)value;  
                Console.WriteLine("{0} ({1}) --> {2} (0x{2:X16}) ({3})",
                                  value, value.GetType().Name,  
                                  lValue, lValue.GetType().Name);  
            }  
            catch (OverflowException)  
            {  
                Console.WriteLine("Unable to convert {0} to Int64.",
                                  value);  
            }  
            try  
            {  
                UInt64 ulValue = (ulong)value;  
                Console.WriteLine("{0} ({1}) --> {2} (0x{2:X16}) ({3})",
                                  value, value.GetType().Name,  
                                  ulValue, ulValue.GetType().Name);  
            }  
            catch (OverflowException)  
            {  
                Console.WriteLine("Unable to convert {0} to UInt64.",
                                  value);  
            }  
            try  
            {  
                Decimal dValue = (decimal)value;  
                Console.WriteLine("{0} ({1}) --> {2} ({3})",
                                  value, value.GetType().Name,  
                                  dValue, dValue.GetType().Name);  
            }  
            catch (OverflowException)  
            {  
                Console.WriteLine("Unable to convert {0} to Decimal.",
                                  value);  
            }  
            try  
            {  
                Single sValue = (float)value;  
                Console.WriteLine("{0} ({1}) --> {2} ({3})",
                                  value, value.GetType().Name,  
                                  sValue, sValue.GetType().Name);  
            }  
            catch (OverflowException)  
            {  
                Console.WriteLine("Unable to convert {0} to Single.",
                                  value);  
            }  
        }  
    }  
}
```

```
value);
        }
        Console.WriteLine();
    }
}
// The example displays the following output for conversions performed
// in a checked context:
//      Unable to convert -1.79769313486232E+308 to Int64.
//      Unable to convert -1.79769313486232E+308 to UInt64.
//      Unable to convert -1.79769313486232E+308 to Decimal.
//      -1.79769313486232E+308 (Double) --> -Infinity (Single)
//
//      -67890.1234 (Double) --> -67890 (0xFFFFFFFFFFFFFEF6CE) (Int64)
//      Unable to convert -67890.1234 to UInt64.
//      -67890.1234 (Double) --> -67890.1234 (Decimal)
//      -67890.1234 (Double) --> -67890.13 (Single)
//
//      -12345.6789 (Double) --> -12345 (0xFFFFFFFFFFFFFCFC7) (Int64)
//      Unable to convert -12345.6789 to UInt64.
//      -12345.6789 (Double) --> -12345.6789 (Decimal)
//      -12345.6789 (Double) --> -12345.68 (Single)
//
//      12345.6789 (Double) --> 12345 (0x0000000000003039) (Int64)
//      12345.6789 (Double) --> 12345 (0x0000000000003039) (UInt64)
//      12345.6789 (Double) --> 12345.6789 (Decimal)
//      12345.6789 (Double) --> 12345.68 (Single)
//
//      67890.1234 (Double) --> 67890 (0x0000000000010932) (Int64)
//      67890.1234 (Double) --> 67890 (0x0000000000010932) (UInt64)
//      67890.1234 (Double) --> 67890.1234 (Decimal)
//      67890.1234 (Double) --> 67890.13 (Single)
//
//      Unable to convert 1.79769313486232E+308 to Int64.
//      Unable to convert 1.79769313486232E+308 to UInt64.
//      Unable to convert 1.79769313486232E+308 to Decimal.
//      1.79769313486232E+308 (Double) --> Infinity (Single)
//
//      Unable to convert NaN to Int64.
//      Unable to convert NaN to UInt64.
//      Unable to convert NaN to Decimal.
//      NaN (Double) --> NaN (Single)
//
//      Unable to convert Infinity to Int64.
//      Unable to convert Infinity to UInt64.
//      Unable to convert Infinity to Decimal.
//      Infinity (Double) --> Infinity (Single)
//
//      Unable to convert -Infinity to Int64.
//      Unable to convert -Infinity to UInt64.
//      Unable to convert -Infinity to Decimal.
//      -Infinity (Double) --> -Infinity (Single)
// The example displays the following output for conversions performed
// in an unchecked context:
```

```
//      -1.79769313486232E+308 (Double) --> -9223372036854775808
(0x8000000000000000) (Int64)
//      -1.79769313486232E+308 (Double) --> 9223372036854775808
(0x8000000000000000) (UInt64)
//      Unable to convert -1.79769313486232E+308 to Decimal.
//      -1.79769313486232E+308 (Double) --> -Infinity (Single)
//
//      -67890.1234 (Double) --> -67890 (0xFFFFFFFFFFFFEF6CE) (Int64)
//      -67890.1234 (Double) --> 18446744073709483726 (0xFFFFFFFFFFFFEF6CE)
(UInt64)
//      -67890.1234 (Double) --> -67890.1234 (Decimal)
//      -67890.1234 (Double) --> -67890.13 (Single)
//
//      -12345.6789 (Double) --> -12345 (0xFFFFFFFFFFFFCFC7) (Int64)
//      -12345.6789 (Double) --> 18446744073709539271 (0xFFFFFFFFFFFFCFC7)
(UInt64)
//      -12345.6789 (Double) --> -12345.6789 (Decimal)
//      -12345.6789 (Double) --> -12345.68 (Single)
//
//      12345.6789 (Double) --> 12345 (0x000000000003039) (Int64)
//      12345.6789 (Double) --> 12345 (0x000000000003039) (UInt64)
//      12345.6789 (Double) --> 12345.6789 (Decimal)
//      12345.6789 (Double) --> 12345.68 (Single)
//
//      67890.1234 (Double) --> 67890 (0x0000000000010932) (Int64)
//      67890.1234 (Double) --> 67890 (0x0000000000010932) (UInt64)
//      67890.1234 (Double) --> 67890.1234 (Decimal)
//      67890.1234 (Double) --> 67890.13 (Single)
//
//      1.79769313486232E+308 (Double) --> -9223372036854775808
(0x8000000000000000) (Int64)
//      1.79769313486232E+308 (Double) --> 0 (0x0000000000000000) (UInt64)
//      Unable to convert 1.79769313486232E+308 to Decimal.
//      1.79769313486232E+308 (Double) --> Infinity (Single)
//
//      NaN (Double) --> -9223372036854775808 (0x8000000000000000) (Int64)
//      NaN (Double) --> 0 (0x0000000000000000) (UInt64)
//      Unable to convert NaN to Decimal.
//      NaN (Double) --> NaN (Single)
//
//      Infinity (Double) --> -9223372036854775808 (0x8000000000000000)
(Int64)
//      Infinity (Double) --> 0 (0x0000000000000000) (UInt64)
//      Unable to convert Infinity to Decimal.
//      Infinity (Double) --> Infinity (Single)
//
//      -Infinity (Double) --> -9223372036854775808 (0x8000000000000000)
(Int64)
//      -Infinity (Double) --> 9223372036854775808 (0x8000000000000000)
(UInt64)
//      Unable to convert -Infinity to Decimal.
//      -Infinity (Double) --> -Infinity (Single)
```

For more information on the conversion of numeric types, see [Type Conversion in .NET](#) and [Type Conversion Tables](#).

Floating-point functionality

The [Double](#) structure and related types provide methods to perform operations in the following areas:

- **Comparison of values.** You can call the [Equals](#) method to determine whether two [Double](#) values are equal, or the [CompareTo](#) method to determine the relationship between two values.

The [Double](#) structure also supports a complete set of comparison operators. For example, you can test for equality or inequality, or determine whether one value is greater than or equal to another. If one of the operands is a numeric type other than a [Double](#), it is converted to a [Double](#) before performing the comparison.

⚠ Warning

Because of differences in precision, two [Double](#) values that you expect to be equal may turn out to be unequal, which affects the result of the comparison. See the [Test for equality](#) section for more information about comparing two [Double](#) values.

You can also call the [IsNaN](#), [IsInfinity](#), [IsPositiveInfinity](#), and [IsNegativeInfinity](#) methods to test for these special values.

- **Mathematical operations.** Common arithmetic operations, such as addition, subtraction, multiplication, and division, are implemented by language compilers and Common Intermediate Language (CIL) instructions, rather than by [Double](#) methods. If one of the operands in a mathematical operation is a numeric type other than a [Double](#), it is converted to a [Double](#) before performing the operation. The result of the operation is also a [Double](#) value.

Other mathematical operations can be performed by calling `static` (`Shared` in Visual Basic) methods in the [System.Math](#) class. It includes additional methods commonly used for arithmetic (such as [Math.Abs](#), [Math.Sign](#), and [Math.Sqrt](#)), geometry (such as [Math.Cos](#) and [Math.Sin](#)), and calculus (such as [Math.Log](#)).

You can also manipulate the individual bits in a [Double](#) value. The [BitConverter.DoubleToInt64Bits](#) method preserves a [Double](#) value's bit pattern in a

64-bit integer. The [BitConverter.GetBytes\(Double\)](#) method returns its bit pattern in a byte array.

- **Rounding.** Rounding is often used as a technique for reducing the impact of differences between values caused by problems of floating-point representation and precision. You can round a [Double](#) value by calling the [Math.Round](#) method.
- **Formatting.** You can convert a [Double](#) value to its string representation by calling the [ToString](#) method or by using the composite formatting feature. For information about how format strings control the string representation of floating-point values, see the [Standard Numeric Format Strings](#) and [Custom Numeric Format Strings](#) topics.
- **Parsing strings.** You can convert the string representation of a floating-point value to a [Double](#) value by calling either the [Parse](#) or [TryParse](#) method. If the parse operation fails, the [Parse](#) method throws an exception, whereas the [TryParse](#) method returns `false`.
- **Type conversion.** The [Double](#) structure provides an explicit interface implementation for the [IConvertible](#) interface, which supports conversion between any two standard .NET Framework data types. Language compilers also support the implicit conversion of values of all other standard numeric types to [Double](#) values. Conversion of a value of any standard numeric type to a [Double](#) is a widening conversion and does not require the user of a casting operator or conversion method,

However, conversion of [Int64](#) and [Single](#) values can involve a loss of precision. The following table lists the differences in precision for each of these types:

[+] Expand table

| Type | Maximum precision | Internal precision |
|------------------------|-------------------|--------------------|
| Double | 15 | 17 |
| Int64 | 19 decimal digits | 19 decimal digits |
| Single | 7 decimal digits | 9 decimal digits |

The problem of precision most frequently affects [Single](#) values that are converted to [Double](#) values. In the following example, two values produced by identical division operations are unequal because one of the values is a single-precision floating point value converted to a [Double](#).

C#

```
using System;

public class Example13
{
    public static void Main()
    {
        Double value = .1;
        Double result1 = value * 10;
        Double result2 = 0;
        for (int ctr = 1; ctr <= 10; ctr++)
            result2 += value;

        Console.WriteLine(".1 * 10: {0:R}", result1);
        Console.WriteLine(".1 Added 10 times: {0:R}", result2);
    }
    // The example displays the following output:
    //      .1 * 10: 1
    //      .1 Added 10 times: 0.9999999999999989
```

Examples

The following code example illustrates the use of [Double](#):

C#

```
// The Temperature class stores the temperature as a Double
// and delegates most of the functionality to the Double
// implementation.
public class Temperature : IComparable, IFormattable
{
    // IComparable.CompareTo implementation.
    public int CompareTo(object obj) {
        if (obj == null) return 1;

        Temperature temp = obj as Temperature;
        if (obj != null)
            return m_value.CompareTo(temp.m_value);
        else
            throw new ArgumentException("object is not a Temperature");
    }

    // IFormattable.ToString implementation.
    public string ToString(string format, IFormatProvider provider) {
        if( format != null ) {
            if( format.Equals("F") ) {
                return String.Format("{0}'F", this.Value.ToString());
            }
            if( format.Equals("C") ) {
```

```
        return String.Format("{0}'C", this.Celsius.ToString()));
    }

    return m_value.ToString(format, provider);
}

// Parses the temperature from a string in the form
// [ws][sign]digits['F'|'C'][ws]
public static Temperature Parse(string s, NumberStyles styles,
IFormatProvider provider) {
    Temperature temp = new Temperature();

    if( s.TrimEnd(null).EndsWith("'F") ) {
        temp.Value = Double.Parse( s.Remove(s.LastIndexOf('\''), 2),
styles, provider);
    }
    else if( s.TrimEnd(null).EndsWith("'C") ) {
        temp.Celsius = Double.Parse( s.Remove(s.LastIndexOf('\''), 2),
styles, provider);
    }
    else {
        temp.Value = Double.Parse(s, styles, provider);
    }

    return temp;
}

// The value holder
protected double m_value;

public double Value {
    get {
        return m_value;
    }
    set {
        m_value = value;
    }
}

public double Celsius {
    get {
        return (m_value-32.0)/1.8;
    }
    set {
        m_value = 1.8*value+32.0;
    }
}
}
```

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

.NET is an open source project.
Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

System.Int32 struct

Article • 01/08/2024

This article provides supplementary remarks to the reference documentation for this API.

[Int32](#) is an immutable value type that represents signed integers with values that range from negative 2,147,483,648 (which is represented by the [Int32.MinValue](#) constant) through positive 2,147,483,647 (which is represented by the [Int32.MaxValue](#) constant). .NET also includes an unsigned 32-bit integer value type, [UInt32](#), which represents values that range from 0 to 4,294,967,295.

Instantiate an Int32 value

You can instantiate an [Int32](#) value in several ways:

- You can declare an [Int32](#) variable and assign it a literal integer value that is within the range of the [Int32](#) data type. The following example declares two [Int32](#) variables and assigns them values in this way.

C#

```
int number1 = 64301;
int number2 = 25548612;
```

- You can assign the value of an integer type whose range is a subset of the [Int32](#) type. This is a widening conversion that does not require a cast operator in C# or a conversion method in Visual Basic but does require one in F#.

C#

```
sbyte value1 = 124;
short value2 = 1618;

int number1 = value1;
int number2 = value2;
```

- You can assign the value of a numeric type whose range exceeds that of the [Int32](#) type. This is a narrowing conversion, so it requires a cast operator in C# or F#, and a conversion method in Visual Basic if [Option Strict](#) is on. If the numeric value is a [Single](#), [Double](#), or [Decimal](#) value that includes a fractional component, the handling of its fractional part depends on the compiler performing the conversion.

The following example performs narrowing conversions to assign several numeric values to [Int32](#) variables.

```
C#  
  
long lNumber = 163245617;  
try {  
    int number1 = (int) lNumber;  
    Console.WriteLine(number1);  
}  
catch (OverflowException) {  
    Console.WriteLine("{0} is out of range of an Int32.", lNumber);  
}  
  
double dbl2 = 35901.997;  
try {  
    int number2 = (int) dbl2;  
    Console.WriteLine(number2);  
}  
catch (OverflowException) {  
    Console.WriteLine("{0} is out of range of an Int32.", dbl2);  
}  
  
BigInteger bigNumber = 132451;  
try {  
    int number3 = (int) bigNumber;  
    Console.WriteLine(number3);  
}  
catch (OverflowException) {  
    Console.WriteLine("{0} is out of range of an Int32.", bigNumber);  
}  
// The example displays the following output:  
//      163245617  
//      35902  
//      132451
```

- You can call a method of the [Convert](#) class to convert any supported type to an [Int32](#) value. This is possible because [Int32](#) supports the [IConvertible](#) interface. The following example illustrates the conversion of an array of [Decimal](#) values to [Int32](#) values.

```
C#  
  
decimal[] values= { Decimal.MinValue, -1034.23m, -12m, 0m, 147m,  
                   199.55m, 9214.16m, Decimal.MaxValue };  
int result;  
  
foreach (decimal value in values)  
{  
    try {  
        result = Convert.ToInt32(value);  
    }
```

```

        Console.WriteLine("Converted the {0} value '{1}' to the {2} value
{3}.",
                           value.GetType().Name, value,
                           result.GetType().Name, result);
    }
    catch (OverflowException) {
        Console.WriteLine("{0} is outside the range of the Int32 type.",
                           value);
    }
}
// The example displays the following output:
//      -79228162514264337593543950335 is outside the range of the Int32
//      type.
//      Converted the Decimal value '-1034.23' to the Int32 value -1034.
//      Converted the Decimal value '-12' to the Int32 value -12.
//      Converted the Decimal value '0' to the Int32 value 0.
//      Converted the Decimal value '147' to the Int32 value 147.
//      Converted the Decimal value '199.55' to the Int32 value 200.
//      Converted the Decimal value '9214.16' to the Int32 value 9214.
//      79228162514264337593543950335 is outside the range of the Int32
//      type.

```

- You can call the [Parse](#) or [TryParse](#) method to convert the string representation of an [Int32](#) value to an [Int32](#). The string can contain either decimal or hexadecimal digits. The following example illustrates the parse operation by using both a decimal and a hexadecimal string.

C#

```

string string1 = "244681";
try {
    int number1 = Int32.Parse(string1);
    Console.WriteLine(number1);
}
catch (OverflowException) {
    Console.WriteLine("{0} is out of range of a 32-bit integer.", string1);
}
catch (FormatException) {
    Console.WriteLine("The format of '{0}' is invalid.", string1);
}

string string2 = "F9A3C";
try {
    int number2 = Int32.Parse(string2,
                               System.Globalization.NumberStyles.HexNumber);
    Console.WriteLine(number2);
}
catch (OverflowException) {
    Console.WriteLine("{0} is out of range of a 32-bit integer.", string2);
}

```

```
        }
        catch (FormatException) {
            Console.WriteLine("The format of '{0}' is invalid.", string2);
        }
        // The example displays the following output:
        //      244681
        //      1022524
```

Perform operations on Int32 values

The [Int32](#) type supports standard mathematical operations such as addition, subtraction, division, multiplication, negation, and unary negation. Like the other integral types, the [Int32](#) type also supports the bitwise `AND`, `OR`, `XOR`, left shift, and right shift operators.

You can use the standard numeric operators to compare two [Int32](#) values, or you can call the [CompareTo](#) or [Equals](#) method.

You can also call the members of the [Math](#) class to perform a wide range of numeric operations, including getting the absolute value of a number, calculating the quotient and remainder from integral division, determining the maximum or minimum value of two integers, getting the sign of a number, and rounding a number.

Represent an Int32 as a string

The [Int32](#) type provides full support for standard and custom numeric format strings. (For more information, see [Formatting Types](#), [Standard Numeric Format Strings](#), and [Custom Numeric Format Strings](#).)

To format an [Int32](#) value as an integral string with no leading zeros, you can call the parameterless [ToString\(\)](#) method. By using the "D" format specifier, you can also include a specified number of leading zeros in the string representation. By using the "N" format specifier, you can include group separators and specify the number of decimal digits to appear in the string representation of the number. By using the "X" format specifier, you can represent an [Int32](#) value as a hexadecimal string. The following example formats the elements in an array of [Int32](#) values in these four ways.

C#

```
int[] numbers = { -1403, 0, 169, 1483104 };
foreach (int number in numbers)
{
    // Display value using default formatting.
    Console.Write("{0,-8} --> ", number.ToString());
    // Display value with 3 digits and leading zeros.
```

```

        Console.WriteLine("{0,11:D3}", number);
        // Display value with 1 decimal digit.
        Console.WriteLine("{0,13:N1}", number);
        // Display value as hexadecimal.
        Console.WriteLine("{0,12:X2}", number);
        // Display value with eight hexadecimal digits.
        Console.WriteLine("{0,14:X8}", number);
    }
    // The example displays the following output:
    //   -1403   -->      -1403      -1,403.0      FFFFFA85      FFFFFA85
    //   0       -->       000        0.0          00      00000000
    //   169     -->       169        169.0        A9      000000A9
    // 1483104   -->    1483104    1,483,104.0     16A160     0016A160

```

You can also format an `Int32` value as a binary, octal, decimal, or hexadecimal string by calling the `ToString(Int32, Int32)` method and supplying the base as the method's second parameter. The following example calls this method to display the binary, octal, and hexadecimal representations of an array of integer values.

C#

```

int[] numbers = { -146, 11043, 2781913 };
Console.WriteLine("{0,8}  {1,32}  {2,11}  {3,10}",
                  "Value", "Binary", "Octal", "Hex");
foreach (int number in numbers)
{
    Console.WriteLine("{0,8}  {1,32}  {2,11}  {3,10}",
                      number, Convert.ToString(number, 2),
                      Convert.ToString(number, 8),
                      Convert.ToString(number, 16));
}
// The example displays the following output:
//   Value           Binary          Octal          Hex
//   -146   111111111111111111111111101101110   37777777556   ffffff6e
//   11043           10101100100011                 25443         2b23
//   2781913         1010100111001011011001       12471331       2a72d9

```

Work with non-decimal 32-bit integer values

In addition to working with individual integers as decimal values, you may want to perform bitwise operations with integer values, or work with the binary or hexadecimal representations of integer values. `Int32` values are represented in 31 bits, with the thirty-second bit used as a sign bit. Positive values are represented by using sign-and-magnitude representation. Negative values are in two's complement representation. This is important to keep in mind when you perform bitwise operations on `Int32` values or when you work with individual bits. In order to perform a numeric, Boolean, or

comparison operation on any two non-decimal values, both values must use the same representation.

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

.NET is an open source project.
Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

System.Int64 struct

Article • 01/08/2024

This article provides supplementary remarks to the reference documentation for this API.

[Int64](#) is an immutable value type that represents signed integers with values that range from negative 9,223,372,036,854,775,808 (which is represented by the [Int64.MinValue](#) constant) through positive 9,223,372,036,854,775,807 (which is represented by the [Int64.MaxValue](#) constant). .NET also includes an unsigned 64-bit integer value type, [UInt64](#), which represents values that range from 0 to 18,446,744,073,709,551,615.

Instantiate an Int64 value

You can instantiate an [Int64](#) value in several ways:

- You can declare an [Int64](#) variable and assign it a literal integer value that is within the range of the [Int64](#) data type. The following example declares two [Int64](#) variables and assigns them values in this way.

C#

```
long number1 = -64301728;
long number2 = 255486129307;
```

- You can assign the value of an integral type whose range is a subset of the [Int64](#) type. This is a widening conversion that does not require a cast operator in C# or a conversion method in Visual Basic. In F#, only the [Int32](#) type can be widened automatically.

C#

```
sbyte value1 = 124;
short value2 = 1618;
int value3 = Int32.MaxValue;

long number1 = value1;
long number2 = value2;
long number3 = value3;
```

- You can assign the value of a numeric type whose range exceeds that of the [Int64](#) type. This is a narrowing conversion, so it requires a cast operator in C# or F# and

a conversion method in Visual Basic if `Option Strict` is on. If the numeric value is a `Single`, `Double`, or `Decimal` value that includes a fractional component, the handling of its fractional part depends on the compiler performing the conversion. The following example performs narrowing conversions to assign several numeric values to `Int64` variables.

C#

```
ulong ulNumber = 163245617943825;
try {
    long number1 = (long) ulNumber;
    Console.WriteLine(number1);
}
catch (OverflowException) {
    Console.WriteLine("{0} is out of range of an Int64.", ulNumber);
}

double dbl2 = 35901.997;
try {
    long number2 = (long) dbl2;
    Console.WriteLine(number2);
}
catch (OverflowException) {
    Console.WriteLine("{0} is out of range of an Int64.", dbl2);
}

BigInteger bigNumber = (BigInteger) 1.63201978555e30;
try {
    long number3 = (long) bigNumber;
    Console.WriteLine(number3);
}
catch (OverflowException) {
    Console.WriteLine("{0} is out of range of an Int64.", bigNumber);
}
// The example displays the following output:
//    163245617943825
//    35902
//    1,632,019,785,549,999,969,612,091,883,520 is out of range of an
Int64.
```

- You can call a method of the `Convert` class to convert any supported type to an `Int64` value. This is possible because `Int64` supports the `IConvertible` interface. The following example illustrates the conversion of an array of `Decimal` values to `Int64` values.

C#

```
decimal[] values= { Decimal.MinValue, -1034.23m, -12m, 0m, 147m,
                    199.55m, 9214.16m, Decimal.MaxValue };
long result;
```

```

foreach (decimal value in values)
{
    try {
        result = Convert.ToInt64(value);
        Console.WriteLine("Converted the {0} value '{1}' to the {2} value
{3}.",
                           value.GetType().Name, value,
                           result.GetType().Name, result);
    }
    catch (OverflowException) {
        Console.WriteLine("{0} is outside the range of the Int64 type.",
                           value);
    }
}
// The example displays the following output:
//      -79228162514264337593543950335 is outside the range of the Int64
type.
//      Converted the Decimal value '-1034.23' to the Int64 value -1034.
//      Converted the Decimal value '-12' to the Int64 value -12.
//      Converted the Decimal value '0' to the Int64 value 0.
//      Converted the Decimal value '147' to the Int64 value 147.
//      Converted the Decimal value '199.55' to the Int64 value 200.
//      Converted the Decimal value '9214.16' to the Int64 value 9214.
//      79228162514264337593543950335 is outside the range of the Int64
type.

```

- You can call the [Parse](#) or [TryParse](#) method to convert the string representation of an [Int64](#) value to an [Int64](#). The string can contain either decimal or hexadecimal digits. The following example illustrates the parse operation by using both a decimal and a hexadecimal string.

C#

```

string string1 = "244681903147";
try {
    long number1 = Int64.Parse(string1);
    Console.WriteLine(number1);
}
catch (OverflowException) {
    Console.WriteLine("'{0}' is out of range of a 64-bit integer.",
string1);
}
catch (FormatException) {
    Console.WriteLine("The format of '{0}' is invalid.", string1);
}

string string2 = "F9A3CFF0A";
try {
    long number2 = Int64.Parse(string2,
System.Globalization.NumberStyles.HexNumber);
}

```

```
        Console.WriteLine(number2);
    }
    catch (OverflowException) {
        Console.WriteLine("{0}' is out of range of a 64-bit integer.", string2);
    }
    catch (FormatException) {
        Console.WriteLine("The format of '{0}' is invalid.", string2);
    }
    // The example displays the following output:
    //      244681903147
    //      67012198154
```

Perform operations on Int64 values

The [Int64](#) type supports standard mathematical operations such as addition, subtraction, division, multiplication, negation, and unary negation. Like the other integral types, the [Int64](#) type also supports the bitwise `AND`, `OR`, `XOR`, left shift, and right shift operators.

You can use the standard numeric operators to compare two [Int64](#) values, or you can call the [CompareTo](#) or [Equals](#) method.

You can also call the members of the [Math](#) class to perform a wide range of numeric operations, including getting the absolute value of a number, calculating the quotient and remainder from integral division, determining the maximum or minimum value of two long integers, getting the sign of a number, and rounding a number.

Represent an Int64 as a string

The [Int64](#) type provides full support for standard and custom numeric format strings. (For more information, see [Formatting Types](#), [Standard Numeric Format Strings](#), and [Custom Numeric Format Strings](#).)

To format an [Int64](#) value as an integral string with no leading zeros, you can call the parameterless [ToString\(\)](#) method. By using the "D" format specifier, you can also include a specified number of leading zeros in the string representation. By using the "N" format specifier, you can include group separators and specify the number of decimal digits to appear in the string representation of the number. By using the "X" format specifier, you can represent an [Int64](#) value as a hexadecimal string. The following example formats the elements in an array of [Int64](#) values in these four ways.

C#

```
long[] numbers = { -1403, 0, 169, 1483104 };
foreach (var number in numbers)
{
    // Display value using default formatting.
    Console.Write("{0,-8} --> ", number.ToString());
    // Display value with 3 digits and leading zeros.
    Console.Write("{0,8:D3}", number);
    // Display value with 1 decimal digit.
    Console.Write("{0,13:N1}", number);
    // Display value as hexadecimal.
    Console.Write("{0,18:X2}", number);
    // Display value with eight hexadecimal digits.
    Console.WriteLine("{0,18:X8}", number);
}
// The example displays the following output:
//      -1403      -->      -1403      -1,403.0  FFFFFFFFFFFFFFA85
FFFFFFFFFFFFFA85
//      0          -->      000          0.0          00
00000000
//      169        -->      169          169.0        A9
000000A9
//      1483104    -->    1483104  1,483,104.0      16A160
0016A160
```

You can also format an `Int64` value as a binary, octal, decimal, or hexadecimal string by calling the `ToString(Int64, Int32)` method and supplying the base as the method's second parameter. The following example calls this method to display the binary, octal, and hexadecimal representations of an array of integer values.

C#

```
// 2781913 (Base 10):  
//     Binary: 1010100111001011011001  
//     Octal: 12471331  
//     Hex: 2a72d9
```

Work with non-decimal 32-bit integer values

In addition to working with individual long integers as decimal values, you may want to perform bitwise operations with long integer values, or work with the binary or hexadecimal representations of long integer values. [Int64](#) values are represented in 63 bits, with the sixty-fourth bit used as a sign bit. Positive values are represented by using sign-and-magnitude representation. Negative values are in two's complement representation. This is important to keep in mind when you perform bitwise operations on [Int64](#) values or when you work with individual bits. In order to perform a numeric, Boolean, or comparison operation on any two non-decimal values, both values must use the same representation.

 Collaborate with us on
[GitHub](#)

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

.NET is an open source project.
Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

System.Numerics.BigInteger struct

Article • 01/08/2024

This article provides supplementary remarks to the reference documentation for this API.

The [BigInteger](#) type is an immutable type that represents an arbitrarily large integer whose value in theory has no upper or lower bounds. The members of the [BigInteger](#) type closely parallel those of other integral types (the [Byte](#), [Int16](#), [Int32](#), [Int64](#), [SByte](#), [UInt16](#), [UInt32](#), and [UInt64](#) types). This type differs from the other integral types in .NET, which have a range indicated by their `MinValue` and `MaxValue` properties.

ⓘ Note

Because the [BigInteger](#) type is immutable (see [Mutability](#)) and because it has no upper or lower bounds, an [OutOfMemoryException](#) can be thrown for any operation that causes a [BigInteger](#) value to grow too large.

Instantiate a BigInteger object

You can instantiate a [BigInteger](#) object in several ways:

- You can use the `new` keyword and provide any integral or floating-point value as a parameter to the [BigInteger](#) constructor. (Floating-point values are truncated before they are assigned to the [BigInteger](#).) The following example illustrates how to use the `new` keyword to instantiate [BigInteger](#) values.

C#

```
BigInteger bigIntFromDouble = new BigInteger(179032.6541);
Console.WriteLine(bigIntFromDouble);
BigInteger bigIntFromInt64 = new BigInteger(934157136952);
Console.WriteLine(bigIntFromInt64);
// The example displays the following output:
// 179032
// 934157136952
```

- You can declare a [BigInteger](#) variable and assign it a value just as you would any numeric type, as long as that value is an integral type. The following example uses assignment to create a [BigInteger](#) value from an [Int64](#).

C#

```
long longValue = 6315489358112;
BigInteger assignedFromLong = longValue;
Console.WriteLine(assignedFromLong);
// The example displays the following output:
// 6315489358112
```

- You can assign a decimal or floating-point value to a `BigInteger` object if you cast the value or convert it first. The following example explicitly casts (in C#) or converts (in Visual Basic) a `Double` and a `Decimal` value to a `BigInteger`.

C#

```
BigInteger assignedFromDouble = (BigInteger) 179032.6541;
Console.WriteLine(assignedFromDouble);
BigInteger assignedFromDecimal = (BigInteger) 64312.65m;
Console.WriteLine(assignedFromDecimal);
// The example displays the following output:
// 179032
// 64312
```

These methods enable you to instantiate a `BigInteger` object whose value is in the range of one of the existing numeric types only. You can instantiate a `BigInteger` object whose value can exceed the range of the existing numeric types in one of three ways:

- You can use the `new` keyword and provide a byte array of any size to the `BigInteger(BigInteger)` constructor. For example:

C#

```
byte[] byteArray = { 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0};
BigInteger newBigInt = new BigInteger(byteArray);
Console.WriteLine("The value of newBigInt is {0} (or 0x{0:x}).",
newBigInt);
// The example displays the following output:
// The value of newBigInt is 4759477275222530853130 (or
0x102030405060708090a).
```

- You can call the `Parse` or `TryParse` methods to convert the string representation of a number to a `BigInteger`. For example:

C#

```
string positiveString = "91389681247993671255432112000000";
string negativeString = "-90315837410896312071002088037140000";
BigInteger posBigInt = 0;
```

```

BigInteger negBigInt = 0;

try {
    posBigInt = BigInteger.Parse(positiveString);
    Console.WriteLine(posBigInt);
}
catch (FormatException)
{
    Console.WriteLine("Unable to convert the string '{0}' to a
BigInteger value.",
                      positiveString);
}

if (BigInteger.TryParse(negativeString, out negBigInt))
    Console.WriteLine(negBigInt);
else
    Console.WriteLine("Unable to convert the string '{0}' to a
BigInteger value.",
                      negativeString);

// The example displays the following output:
//   9.1389681247993671255432112E+31
//   -9.0315837410896312071002088037E+34

```

- You can call a `static` (`Shared` in Visual Basic) `BigInteger` method that performs some operation on a numeric expression and returns a calculated `BigInteger` result. The following example does this by cubing `UInt64.MaxValue` and assigning the result to a `BigInteger`.

C#

```

BigInteger number = BigInteger.Pow(UInt64.MaxValue, 3);
Console.WriteLine(number);
// The example displays the following output:
//   627710173538668076281494232244851025767571854389858533375

```

The uninitialized value of a `BigInteger` is `Zero`.

Perform operations on `BigInteger` values

You can use a `BigInteger` instance as you would use any other integral type. `BigInteger` overloads the standard numeric operators to enable you to perform basic mathematical operations such as addition, subtraction, division, multiplication, and unary negation. You can also use the standard numeric operators to compare two `BigInteger` values with each other. Like the other integral types, `BigInteger` also supports the bitwise `And`, `Or`, `Xor`, left shift, and right shift operators. For languages that do not support custom operators, the `BigInteger` structure also provides equivalent methods for performing

mathematical operations. These include [Add](#), [Divide](#), [Multiply](#), [Negate](#), [Subtract](#), and several others.

Many members of the [BigInteger](#) structure correspond directly to members of the other integral types. In addition, [BigInteger](#) adds members such as the following:

- [Sign](#), which returns a value that indicates the sign of a [BigInteger](#) value.
- [Abs](#), which returns the absolute value of a [BigInteger](#) value.
- [DivRem](#), which returns both the quotient and remainder of a division operation.
- [GreatestCommonDivisor](#), which returns the greatest common divisor of two [BigInteger](#) values.

Many of these additional members correspond to the members of the [Math](#) class, which provides the functionality to work with the primitive numeric types.

Mutability

The following example instantiates a [BigInteger](#) object and then increments its value by one.

```
C#
```

```
BigInteger number = BigInteger.Multiply(Int64.MaxValue, 3);
number++;
Console.WriteLine(number);
```

Although this example appears to modify the value of the existing object, this is not the case. [BigInteger](#) objects are immutable, which means that internally, the common language runtime actually creates a new [BigInteger](#) object and assigns it a value one greater than its previous value. This new object is then returned to the caller.

ⓘ Note

The other numeric types in .NET are also immutable. However, because the [BigInteger](#) type has no upper or lower bounds, its values can grow extremely large and have a measurable impact on performance.

Although this process is transparent to the caller, it does incur a performance penalty. In some cases, especially when repeated operations are performed in a loop on very large [BigInteger](#) values, that performance penalty can be significant. For example, in the

following example, an operation is performed repetitively up to a million times, and a [BigInteger](#) value is incremented by one every time the operation succeeds.

C#

```
BigInteger number = Int64.MaxValue ^ 5;
int repetitions = 1000000;
// Perform some repetitive operation 1 million times.
for (int ctr = 0; ctr <= repetitions; ctr++)
{
    // Perform some operation. If it fails, exit the loop.
    if (!SomeOperationSucceeds()) break;
    // The following code executes if the operation succeeds.
    number++;
}
```

In such a case, you can improve performance by performing all intermediate assignments to an [Int32](#) variable. The final value of the variable can then be assigned to the [BigInteger](#) object when the loop exits. The following example provides an illustration.

C#

```
BigInteger number = Int64.MaxValue ^ 5;
int repetitions = 1000000;
int actualRepetitions = 0;
// Perform some repetitive operation 1 million times.
for (int ctr = 0; ctr <= repetitions; ctr++)
{
    // Perform some operation. If it fails, exit the loop.
    if (!SomeOperationSucceeds()) break;
    // The following code executes if the operation succeeds.
    actualRepetitions++;
}
number += actualRepetitions;
```

Byte arrays and hexadecimal strings

If you convert [BigInteger](#) values to byte arrays, or if you convert byte arrays to [BigInteger](#) values, you must consider the order of bytes. The [BigInteger](#) structure expects the individual bytes in a byte array to appear in little-endian order (that is, the lower-order bytes of the value precede the higher-order bytes). You can round-trip a [BigInteger](#) value by calling the [ToByteArray](#) method and then passing the resulting byte array to the [BigInteger\(Byte\[\]\)](#) constructor, as the following example shows.

C#

```

BigInteger number = BigInteger.Pow(Int64.MaxValue, 2);
Console.WriteLine(number);

// Write the BigInteger value to a byte array.
byte[] bytes = number.ToByteArray();

// Display the byte array.
foreach (byte byteValue in bytes)
    Console.Write("0x{0:X2} ", byteValue);
Console.WriteLine();

// Restore the BigInteger value from a Byte array.
BigInteger newNumber = new BigInteger(bytes);
Console.WriteLine(newNumber);
// The example displays the following output:
//     8.5070591730234615847396907784E+37
//     0x01 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0xFF 0xFF 0xFF 0xFF 0xFF 0xFF
0xFF 0x3F
//
//     8.5070591730234615847396907784E+37

```

To instantiate a [BigInteger](#) value from a byte array that represents a value of some other integral type, you can pass the integral value to the [BitConverter.GetBytes](#) method, and then pass the resulting byte array to the [BigInteger\(Byte\[\]\)](#) constructor. The following example instantiates a [BigInteger](#) value from a byte array that represents an [Int16](#) value.

C#

```

short originalValue = 30000;
Console.WriteLine(originalValue);

// Convert the Int16 value to a byte array.
byte[] bytes = BitConverter.GetBytes(originalValue);

// Display the byte array.
foreach (byte byteValue in bytes)
    Console.Write("0x{0} ", byteValue.ToString("X2"));
Console.WriteLine();

// Pass byte array to the BigInteger constructor.
BigInteger number = new BigInteger(bytes);
Console.WriteLine(number);
// The example displays the following output:
//     30000
//     0x30 0x75
//     30000

```

The [BigInteger](#) structure assumes that negative values are stored by using two's complement representation. Because the [BigInteger](#) structure represents a numeric

value with no fixed length, the [BigInteger\(Byte\[\]\)](#) constructor always interprets the most significant bit of the last byte in the array as a sign bit. To prevent the [BigInteger\(Byte\[\]\)](#) constructor from confusing the two's complement representation of a negative value with the sign and magnitude representation of a positive value, positive values in which the most significant bit of the last byte in the byte array would ordinarily be set should include an additional byte whose value is 0. For example, 0xC0 0xBD 0xF0 0xFF is the little-endian hexadecimal representation of either -1,000,000 or 4,293,967,296. Because the most significant bit of the last byte in this array is on, the value of the byte array would be interpreted by the [BigInteger\(Byte\[\]\)](#) constructor as -1,000,000. To instantiate a [BigInteger](#) whose value is positive, a byte array whose elements are 0xC0 0xBD 0xF0 0xFF 0x00 must be passed to the constructor. The following example illustrates this.

C#

```
int negativeNumber = -1000000;
uint positiveNumber = 4293967296;

byte[] negativeBytes = BitConverter.GetBytes(negativeNumber);
BigInteger negativeBigInt = new BigInteger(negativeBytes);
Console.WriteLine(negativeBigInt.ToString("N0"));

byte[] tempPosBytes = BitConverter.GetBytes(positiveNumber);
byte[] positiveBytes = new byte[tempPosBytes.Length + 1];
Array.Copy(tempPosBytes, positiveBytes, tempPosBytes.Length);
BigInteger positiveBigInt = new BigInteger(positiveBytes);
Console.WriteLine(positiveBigInt.ToString("N0"));
// The example displays the following output:
//      -1,000,000
//      4,293,967,296
```

Byte arrays created by the [ToByteArray](#) method from positive values include this extra zero-value byte. Therefore, the [BigInteger](#) structure can successfully round-trip values by assigning them to, and then restoring them from, byte arrays, as the following example shows.

C#

```
BigInteger positiveValue = 15777216;
BigInteger negativeValue = -1000000;

Console.WriteLine("Positive value: " + positiveValue.ToString("N0"));
byte[] bytes = positiveValue.ToByteArray();

foreach (byte byteValue in bytes)
    Console.Write("{0:X2} ", byteValue);
Console.WriteLine();
positiveValue = new BigInteger(bytes);
Console.WriteLine("Restored positive value: " +
```

```

positiveValue.ToString("N0"));

Console.WriteLine();

Console.WriteLine("Negative value: " + negativeValue.ToString("N0"));
bytes = negativeValue.ToByteArray();
foreach (byte byteValue in bytes)
    Console.Write("{0:X2} ", byteValue);
Console.WriteLine();
negativeValue = new BigInteger(bytes);
Console.WriteLine("Restored negative value: " +
negativeValue.ToString("N0"));
// The example displays the following output:
//      Positive value: 15,777,216
//      C0 BD F0 00
//      Restored positive value: 15,777,216
//
//      Negative value: -1,000,000
//      C0 BD F0
//      Restored negative value: -1,000,000

```

However, you may need to add this additional zero-value byte to byte arrays that are created dynamically by the developer or that are returned by methods that convert unsigned integers to byte arrays (such as [BitConverter.GetBytes\(UInt16\)](#), [BitConverter.GetBytes\(UInt32\)](#), and [BitConverter.GetBytes\(UInt64\)](#)).

When parsing a hexadecimal string, the [BigInteger.Parse\(String, NumberStyles\)](#) and [BigInteger.Parse\(String, NumberStyles, IFormatProvider\)](#) methods assume that if the most significant bit of the first byte in the string is set, or if the first hexadecimal digit of the string represents the lower four bits of a byte value, the value is represented by using two's complement representation. For example, both "FF01" and "F01" represent the decimal value -255. To differentiate positive from negative values, positive values should include a leading zero. The relevant overloads of the [ToString](#) method, when they are passed the "X" format string, add a leading zero to the returned hexadecimal string for positive values. This makes it possible to round-trip [BigInteger](#) values by using the [ToString](#) and [Parse](#) methods, as the following example shows.

C#

```

BigInteger negativeNumber = -1000000;
BigInteger positiveNumber = 15777216;

string negativeHex = negativeNumber.ToString("X");
string positiveHex = positiveNumber.ToString("X");

BigInteger negativeNumber2, positiveNumber2;
negativeNumber2 = BigInteger.Parse(negativeHex,
                                   NumberStyles.HexNumber);
positiveNumber2 = BigInteger.Parse(positiveHex,

```

```

        NumberStyles.HexNumber);

Console.WriteLine("Converted {0:N0} to {1} back to {2:N0}.",
                  negativeNumber, negativeHex, negativeNumber2);
Console.WriteLine("Converted {0:N0} to {1} back to {2:N0}.",
                  positiveNumber, positiveHex, positiveNumber2);
// The example displays the following output:
//      Converted -1,000,000 to F0BDC0 back to -1,000,000.
//      Converted 15,777,216 to 0F0BDC0 back to 15,777,216.

```

However, the hexadecimal strings created by calling the `ToString` methods of the other integral types or the overloads of the `ToString` method that include a `toBase` parameter do not indicate the sign of the value or the source data type from which the hexadecimal string was derived. Successfully instantiating a `BigInteger` value from such a string requires some additional logic. The following example provides one possible implementation.

C#

```

using System;
using System.Globalization;
using System.Numerics;

public struct HexValue
{
    public int Sign;
    public string Value;
}

public class ByteHexExample2
{
    public static void Main()
    {
        uint positiveNumber = 4039543321;
        int negativeNumber = -255423975;

        // Convert the numbers to hex strings.
        HexValue hexValue1, hexValue2;
        hexValue1.Value = positiveNumber.ToString("X");
        hexValue1.Sign = Math.Sign(positiveNumber);

        hexValue2.Value = Convert.ToString(negativeNumber, 16);
        hexValue2.Sign = Math.Sign(negativeNumber);

        // Round-trip the hexadecimal values to BigInteger values.
        string hexString;
        BigInteger positiveBigInt, negativeBigInt;

        hexString = (hexValue1.Sign == 1 ? "0" : "") + hexValue1.Value;
        positiveBigInt = BigInteger.Parse(hexString,
NumberStyles.HexNumber);

```

```
        Console.WriteLine("Converted {0} to {1} and back to {2}.",
                           positiveNumber, hexValue1.Value, positiveBigInt);

        hexString = (hexValue2.Sign == 1 ? "0" : "") + hexValue2.Value;
        negativeBigInt = BigInteger.Parse(hexString,
NumberStyles.HexNumber);
        Console.WriteLine("Converted {0} to {1} and back to {2}.",
                           negativeNumber, hexValue2.Value, negativeBigInt);
    }
}

// The example displays the following output:
//      Converted 4039543321 to F0C68A19 and back to 4039543321.
//      Converted -255423975 to f0c68a19 and back to -255423975.
```

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

.NET is an open source project.
Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

System.Numerics.Complex struct

Article • 01/08/2024

This article provides supplementary remarks to the reference documentation for this API.

A complex number is a number that comprises a real number part and an imaginary number part. A complex number z is usually written in the form $z = x + yi$, where x and y are real numbers, and i is the imaginary unit that has the property $i^2 = -1$. The real part of the complex number is represented by x , and the imaginary part of the complex number is represented by y .

The [Complex](#) type uses the Cartesian coordinate system (real, imaginary) when instantiating and manipulating complex numbers. A complex number can be represented as a point in a two-dimensional coordinate system, which is known as the complex plane. The real part of the complex number is positioned on the x-axis (the horizontal axis), and the imaginary part is positioned on the y-axis (the vertical axis).

Any point in the complex plane can also be expressed based on its absolute value, by using the polar coordinate system. In polar coordinates, a point is characterized by two numbers:

- Its magnitude, which is the distance of the point from the origin (that is, 0,0, or the point at which the x-axis and the y-axis intersect).
- Its phase, which is the angle between the real axis and the line drawn from the origin to the point.

Instantiate a complex number

You can assign a value to a complex number in one of the following ways:

- By passing two [Double](#) values to its constructor. The first value represents the real part of the complex number, and the second value represents its imaginary part. These values represent the position of the complex number in the two-dimensional Cartesian coordinate system.
- By calling the static (`Shared` in Visual Basic) [Complex.FromPolarCoordinates](#) method to create a complex number from its polar coordinates.
- By assigning a [Byte](#), [SByte](#), [Int16](#), [UInt16](#), [Int32](#), [UInt32](#), [Int64](#), [UInt64](#), [Single](#), or [Double](#) value to a [Complex](#) object. The value becomes the real part of the complex number, and its imaginary part equals 0.

- By casting (in C#) or converting (in Visual Basic) a [Decimal](#) or [BigInteger](#) value to a [Complex](#) object. The value becomes the real part of the complex number, and its imaginary part equals 0.
- By assigning the complex number that is returned by a method or operator to a [Complex](#) object. For example, [Complex.Add](#) is a static method that returns a complex number that is the sum of two complex numbers, and the [Complex.Addition](#) operator adds two complex numbers and returns the result.

The following example demonstrates each of these five ways of assigning a value to a complex number.

C#

```
using System;
using System.Numerics;

public class CreateEx
{
    public static void Main()
    {
        // Create a complex number by calling its class constructor.
        Complex c1 = new Complex(12, 6);
        Console.WriteLine(c1);

        // Assign a Double to a complex number.
        Complex c2 = 3.14;
        Console.WriteLine(c2);

        // Cast a Decimal to a complex number.
        Complex c3 = (Complex)12.3m;
        Console.WriteLine(c3);

        // Assign the return value of a method to a Complex variable.
        Complex c4 = Complex.Pow(Complex.One, -1);
        Console.WriteLine(c4);

        // Assign the value returned by an operator to a Complex variable.
        Complex c5 = Complex.One + Complex.One;
        Console.WriteLine(c5);

        // Instantiate a complex number from its polar coordinates.
        Complex c6 = Complex.FromPolarCoordinates(10, .524);
        Console.WriteLine(c6);
    }
}

// The example displays the following output:
//      (12, 6)
//      (3.14, 0)
//      (12.3, 0)
//      (1, 0)
```

```
//      (2, 0)
//      (8.65824721882145, 5.00347430269914)
```

Operations with complex numbers

The [Complex](#) structure in .NET includes members that provide the following functionality:

- Methods to compare two complex numbers to determine whether they are equal.
- Operators to perform arithmetic operations on complex numbers. [Complex](#) operators enable you to perform addition, subtraction, multiplication, division, and unary negation with complex numbers.
- Methods to perform other numerical operations on complex numbers. In addition to the four basic arithmetic operations, you can raise a complex number to a specified power, find the square root of a complex number, and get the absolute value of a complex number.
- Methods to perform trigonometric operations on complex numbers. For example, you can calculate the tangent of an angle represented by a complex number.

Note that, because the [Real](#) and [Imaginary](#) properties are read-only, you cannot modify the value of an existing [Complex](#) object. All methods that perform an operation on a [Complex](#) number, if their return value is of type [Complex](#), return a new [Complex](#) number.

Precision and complex numbers

The real and imaginary parts of a complex number are represented by two double-precision floating-point values. This means that [Complex](#) values, like double-precision floating-point values, can lose precision as a result of numerical operations. This means that strict comparisons for equality of two [Complex](#) values may fail, even if the difference between the two values is due to a loss of precision. For more information, see [Double](#).

For example, performing exponentiation on the logarithm of a number should return the original number. However, in some cases, the loss of precision of floating-point values can cause slight differences between the two values, as the following example illustrates.

C#

```
Complex value = new Complex(Double.MinValue / 2, Double.MinValue / 2);
Complex value2 = Complex.Exp(Complex.Log(value));
```

```

Console.WriteLine("{0} \n{1} \nEqual: {2}", value, value2,
                  value == value2);
// The example displays the following output:
//      (-8.98846567431158E+307, -8.98846567431158E+307)
//      (-8.98846567431161E+307, -8.98846567431161E+307)
//      Equal: False

```

Similarly, the following example, which calculates the square root of a [Complex](#) number, produces slightly different results on the 32-bit and IA64 versions of .NET.

C#

```

Complex minusOne = new Complex(-1, 0);
Console.WriteLine(Complex.Sqrt(minusOne));
// The example displays the following output:
//      (6.12303176911189E-17, 1) on 32-bit systems.
//      (6.12323399573677E-17,1) on IA64 systems.

```

Infinity and NaN

The real and imaginary parts of a complex number are represented by [Double](#) values. In addition to ranging from [Double.MinValue](#) to [Double.MaxValue](#), the real or imaginary part of a complex number can have a value of [Double.PositiveInfinity](#), [Double.NegativeInfinity](#), or [Double.NaN](#). [Double.PositiveInfinity](#), [Double.NegativeInfinity](#), and [Double.NaN](#) all propagate in any arithmetic or trigonometric operation.

In the following example, division by [Zero](#) produces a complex number whose real and imaginary parts are both [Double.NaN](#). As a result, performing multiplication with this value also produces a complex number whose real and imaginary parts are [Double.NaN](#). Similarly, performing a multiplication that overflows the range of the [Double](#) type produces a complex number whose real part is [Double.NaN](#) and whose imaginary part is [Double.PositiveInfinity](#). Subsequently performing division with this complex number returns a complex number whose real part is [Double.NaN](#) and whose imaginary part is [Double.PositiveInfinity](#).

C#

```

using System;
using System.Numerics;

public class NaNEx
{
    public static void Main()
    {
        Complex c1 = new Complex(Double.MaxValue / 2, Double.MaxValue / 2);

```

```

        Complex c2 = c1 / Complex.Zero;
        Console.WriteLine(c2.ToString());
        c2 = c2 * new Complex(1.5, 1.5);
        Console.WriteLine(c2.ToString());
        Console.WriteLine();

        Complex c3 = c1 * new Complex(2.5, 3.5);
        Console.WriteLine(c3.ToString());
        c3 = c3 + new Complex(Double.MinValue / 2, Double.MaxValue / 2);
        Console.WriteLine(c3);
    }

// The example displays the following output:
//      (NaN, NaN)
//      (NaN, NaN)
//      (NaN, Infinity)
//      (NaN, Infinity)

```

Mathematical operations with complex numbers that are invalid or that overflow the range of the [Double](#) data type do not throw an exception. Instead, they return a [Double.PositiveInfinity](#), [Double.NegativeInfinity](#), or [Double.NaN](#) under the following conditions:

- The division of a positive number by zero returns [Double.PositiveInfinity](#).
- Any operation that overflows the upper bound of the [Double](#) data type returns [Double.PositiveInfinity](#).
- The division of a negative number by zero returns [Double.NegativeInfinity](#).
- Any operation that overflows the lower bound of the [Double](#) data type returns [Double.NegativeInfinity](#).
- The division of a zero by zero returns [Double.NaN](#).
- Any operation that is performed on operands whose values are [Double.PositiveInfinity](#), [Double.NegativeInfinity](#), or [Double.NaN](#) returns [Double.PositiveInfinity](#), [Double.NegativeInfinity](#), or [Double.NaN](#), depending on the specific operation.

Note that this applies to any intermediate calculations performed by a method. For example, the multiplication of `new Complex(9e308, 9e308)` and `new Complex(2.5, 3.5)` uses the formula $(ac - bd) + (ad + bc)i$. The calculation of the real component that results from the multiplication evaluates the expression $9e308 \cdot 2.5 - 9e308 \cdot 3.5$. Each intermediate multiplication in this expression returns [Double.PositiveInfinity](#), and the attempt to subtract [Double.PositiveInfinity](#) from [Double.PositiveInfinity](#) returns [Double.NaN](#).

Format a complex number

By default, the string representation of a complex number takes the form `(real, imaginary)`, where *real* and *imaginary* are the string representations of the `Double` values that form the complex number's real and imaginary components. Some overloads of the `ToString` method allow customization of the string representations of these `Double` values to reflect the formatting conventions of a particular culture or to appear in a particular format defined by a standard or custom numeric format string. (For more information, see [Standard Numeric Format Strings](#) and [Custom Numeric Format Strings](#).)

One of the more common ways of expressing the string representation of a complex number takes the form $a + bi$, where a is the complex number's real component, and b is the complex number's imaginary component. In electrical engineering, a complex number is most commonly expressed as $a + bj$. You can return the string representation of a complex number in either of these two forms. To do this, define a custom format provider by implementing the `ICustomFormatter` and `IFormatProvider` interfaces, and then call the `String.Format(IFormatProvider, String, Object[])` method.

The following example defines a `ComplexFormatter` class that represents a complex number as a string in the form of either $a + bi$ or $a + bj$.

C#

```
using System;
using System.Numerics;

public class ComplexFormatter : IFormatProvider, ICustomFormatter
{
    public object GetFormat(Type formatType)
    {
        if (formatType == typeof(ICustomFormatter))
            return this;
        else
            return null;
    }

    public string Format(string format, object arg,
                         IFormatProvider provider)
    {
        if (arg is Complex)
        {
            Complex c1 = (Complex)arg;
            // Check if the format string has a precision specifier.
            int precision;
            string fmtString = String.Empty;
            if (format.Length > 1)
            {
                try
                {

```

```

        precision = Int32.Parse(format.Substring(1));
    }
    catch (FormatException)
    {
        precision = 0;
    }
    fmtString = "N" + precision.ToString();
}
if (format.Substring(0, 1).Equals("I",
 StringComparison.OrdinalIgnoreCase))
    return c1.Real.ToString(fmtString) + " + " +
c1.Imaginary.ToString(fmtString) + "i";
else if (format.Substring(0, 1).Equals("J",
 StringComparison.OrdinalIgnoreCase))
    return c1.Real.ToString(fmtString) + " + " +
c1.Imaginary.ToString(fmtString) + "j";
else
    return c1.ToString(format, provider);
}
else
{
    if (arg is IFormattable)
        return ((IFormattable)arg).ToString(format, provider);
    else if (arg != null)
        return arg.ToString();
    else
        return String.Empty;
}
}
}

```

The following example then uses this custom formatter to display the string representation of a complex number.

C#

```

public class CustomFormatEx
{
    public static void Main()
    {
        Complex c1 = new Complex(12.1, 15.4);
        Console.WriteLine("Formatting with ToString():      " +
                          c1.ToString());
        Console.WriteLine("Formatting with ToString(format): " +
                          c1.ToString("N2"));
        Console.WriteLine("Custom formatting with I0:      " +
                          String.Format(new ComplexFormatter(), "{0:I0}",
c1));
        Console.WriteLine("Custom formatting with J3:      " +
                          String.Format(new ComplexFormatter(), "{0:J3}",
c1));
    }
}

```

```
// The example displays the following output:  
//   Formatting with ToString(): (12.1, 15.4)  
//   Formatting with ToString(format): (12.10, 15.40)  
//   Custom formatting with I0: 12 + 15i  
//   Custom formatting with J3: 12.100 + 15.400j
```

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

.NET is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

System.Single struct

Article • 01/08/2024

This article provides supplementary remarks to the reference documentation for this API.

The [Single](#) value type represents a single-precision 32-bit number with values ranging from negative 3.402823e38 to positive 3.402823e38, as well as positive or negative zero, [PositiveInfinity](#), [NegativeInfinity](#), and not a number ([NaN](#)). It is intended to represent values that are extremely large (such as distances between planets or galaxies) or extremely small (such as the molecular mass of a substance in kilograms) and that often are imprecise (such as the distance from earth to another solar system). The [Single](#) type complies with the IEC 60559:1989 (IEEE 754) standard for binary floating-point arithmetic.

[System.Single](#) provides methods to compare instances of this type, to convert the value of an instance to its string representation, and to convert the string representation of a number to an instance of this type. For information about how format specification codes control the string representation of value types, see [Formatting Types](#), [Standard Numeric Format Strings](#), and [Custom Numeric Format Strings](#).

Floating-point representation and precision

The [Single](#) data type stores single-precision floating-point values in a 32-bit binary format, as shown in the following table:

[Expand table](#)

| Part | Bits |
|-----------------------------------|-------|
| Significand or mantissa | 0-22 |
| Exponent | 23-30 |
| Sign (0 = positive, 1 = negative) | 31 |

Just as decimal fractions are unable to precisely represent some fractional values (such as 1/3 or [Math.PI](#)), binary fractions are unable to represent some fractional values. For example, 2/10, which is represented precisely by .2 as a decimal fraction, is represented by .0011111001001100 as a binary fraction, with the pattern "1100" repeating to infinity. In this case, the floating-point value provides an imprecise representation of the number that it represents. Performing additional mathematical operations on the original

floating-point value often increases its lack of precision. For example, if you compare the results of multiplying .3 by 10 and adding .3 to .3 nine times, you will see that addition produces the less precise result, because it involves eight more operations than multiplication. Note that this disparity is apparent only if you display the two [Single](#) values by using the "R" [standard numeric format string](#), which, if necessary, displays all 9 digits of precision supported by the [Single](#) type.

C#

```
using System;

public class Example12
{
    public static void Main()
    {
        Single value = .2f;
        Single result1 = value * 10f;
        Single result2 = 0f;
        for (int ctr = 1; ctr <= 10; ctr++)
            result2 += value;

        Console.WriteLine(".2 * 10: {0:R}", result1);
        Console.WriteLine(".2 Added 10 times: {0:R}", result2);
    }
}

// The example displays the following output:
//      .2 * 10: 2
//      .2 Added 10 times: 2.00000024
```

Because some numbers cannot be represented exactly as fractional binary values, floating-point numbers can only approximate real numbers.

All floating-point numbers have a limited number of significant digits, which also determines how accurately a floating-point value approximates a real number. A [Single](#) value has up to 7 decimal digits of precision, although a maximum of 9 digits is maintained internally. This means that some floating-point operations may lack the precision to change a floating-point value. The following example defines a large single-precision floating-point value, and then adds the product of [Single.Epsilon](#) and one quadrillion to it. However, the product is too small to modify the original floating-point value. Its least significant digit is thousandths, whereas the most significant digit in the product is 10^{-30} .

C#

```
using System;

public class Example13
```

```

{
    public static void Main()
    {
        Single value = 123.456f;
        Single additional = Single.Epsilon * 1e15f;
        Console.WriteLine($"{value} + {additional} = {value + additional}");
    }
}
// The example displays the following output:
//      123.456 + 1.401298E-30 = 123.456

```

The limited precision of a floating-point number has several consequences:

- Two floating-point numbers that appear equal for a particular precision might not compare equal because their least significant digits are different. In the following example, a series of numbers are added together, and their total is compared with their expected total. Although the two values appear to be the same, a call to the `Equals` method indicates that they are not.

C#

```

using System;

public class Example9
{
    public static void Main()
    {
        Single[] values = { 10.01f, 2.88f, 2.88f, 2.88f, 9.0f };
        Single result = 27.65f;
        Single total = 0f;
        foreach (var value in values)
            total += value;

        if (total.Equals(result))
            Console.WriteLine("The sum of the values equals the
total.");
        else
            Console.WriteLine("The sum of the values ({0}) does not
equal the total ({1}).",
                            total, result);
    }
}
// The example displays the following output:
//      The sum of the values (27.65) does not equal the total (27.65).
//
// If the index items in the Console.WriteLine statement are changed to
// {0:R},
// the example displays the following output:
//      The sum of the values (27.6500015) does not equal the total
//(27.65).

```

If you change the format items in the `Console.WriteLine(String, Object, Object)` statement from `{0}` and `{1}` to `{0:R}` and `{1:R}` to display all significant digits of the two `Single` values, it is clear that the two values are unequal because of a loss of precision during the addition operations. In this case, the issue can be resolved by calling the `Math.Round(Double, Int32)` method to round the `Single` values to the desired precision before performing the comparison.

- A mathematical or comparison operation that uses a floating-point number might not yield the same result if a decimal number is used, because the binary floating-point number might not equal the decimal number. A previous example illustrated this by displaying the result of multiplying .3 by 10 and adding .3 to .3 nine times.

When accuracy in numeric operations with fractional values is important, use the `Decimal` type instead of the `Single` type. When accuracy in numeric operations with integral values beyond the range of the `Int64` or `UInt64` types is important, use the `BigInteger` type.

- A value might not round-trip if a floating-point number is involved. A value is said to round-trip if an operation converts an original floating-point number to another form, an inverse operation transforms the converted form back to a floating-point number, and the final floating-point number is equal to the original floating-point number. The round trip might fail because one or more least significant digits are lost or changed in a conversion. In the following example, three `Single` values are converted to strings and saved in a file. As the output shows, although the values appear to be identical, the restored values are not equal to the original values.

C#

```
using System;
using System.IO;

public class Example10
{
    public static void Main()
    {
        StreamWriter sw = new StreamWriter(@".\Singles.dat");
        Single[] values = { 3.2f / 1.11f, 1.0f / 3f, (float)Math.PI };
        for (int ctr = 0; ctr < values.Length; ctr++)
        {
            sw.Write(values[ctr].ToString());
            if (ctr != values.Length - 1)
                sw.Write("|");
        }
        sw.Close();

        Single[] restoredValues = new Single[values.Length];
        StreamReader sr = new StreamReader(@".\Singles.dat");
```

```

        string temp = sr.ReadToEnd();
        string[] tempStrings = temp.Split(' ');
        for (int ctr = 0; ctr < tempStrings.Length; ctr++)
            restoredValues[ctr] = Single.Parse(tempStrings[ctr]);

        for (int ctr = 0; ctr < values.Length; ctr++)
            Console.WriteLine("{0} {2} {1}", values[ctr],
                restoredValues[ctr],
                values[ctr].Equals(restoredValues[ctr]) ?
                    "=" : "<>");
    }
}

// The example displays the following output:
//      2.882883 <> 2.882883
//      0.3333333 <> 0.3333333
//      3.141593 <> 3.141593

```

In this case, the values can be successfully round-tripped by using the "G9" standard numeric format string to preserve the full precision of `Single` values, as the following example shows.

C#

```

using System;
using System.IO;

public class Example11
{
    public static void Main()
    {
        StreamWriter sw = new StreamWriter(@".\Singles.dat");
        Single[] values = { 3.2f / 1.11f, 1.0f / 3f, (float)Math.PI };
        for (int ctr = 0; ctr < values.Length; ctr++)
            sw.Write("{0:G9}{1}", values[ctr], ctr < values.Length - 1
? " | " : "");

        sw.Close();

        Single[] restoredValues = new Single[values.Length];
        StreamReader sr = new StreamReader(@".\Singles.dat");
        string temp = sr.ReadToEnd();
        string[] tempStrings = temp.Split(' ');
        for (int ctr = 0; ctr < tempStrings.Length; ctr++)
            restoredValues[ctr] = Single.Parse(tempStrings[ctr]);

        for (int ctr = 0; ctr < values.Length; ctr++)
            Console.WriteLine("{0} {2} {1}", values[ctr],
                restoredValues[ctr],
                values[ctr].Equals(restoredValues[ctr]) ?
                    "=" : "<>");
    }
}

```

```
// The example displays the following output:  
//      2.882883 = 2.882883  
//      0.3333333 = 0.3333333  
//      3.141593 = 3.141593
```

- Single values have less precision than Double values. A Single value that is converted to a seemingly equivalent Double often does not equal the Double value because of differences in precision. In the following example, the result of identical division operations is assigned to a Double value and a Single value. After the Single value is cast to a Double, a comparison of the two values shows that they are unequal.

C#

```
using System;  
  
public class Example9  
{  
    public static void Main()  
    {  
        Double value1 = 1 / 3.0;  
        Single sValue2 = 1 / 3.0f;  
        Double value2 = (Double)sValue2;  
        Console.WriteLine("{0:R} = {1:R}: {2}", value1, value2,  
                           value1.Equals(value2));  
    }  
}  
// The example displays the following output:  
//      0.3333333333333331 = 0.333333432674408: False
```

To avoid this problem, either use the Double data type in place of the Single data type, or use the Round method so that both values have the same precision.

Test for equality

To be considered equal, two Single values must represent identical values. However, because of differences in precision between values, or because of a loss of precision by one or both values, floating-point values that are expected to be identical often turn out to be unequal due to differences in their least significant digits. As a result, calls to the Equals method to determine whether two values are equal, or calls to the CompareTo method to determine the relationship between two Single values, often yield unexpected results. This is evident in the following example, where two apparently equal Single values turn out to be unequal, because the first value has 7 digits of precision, whereas the second value has 9.

C#

```
using System;

public class Example
{
    public static void Main()
    {
        float value1 = .3333333f;
        float value2 = 1.0f/3;
        Console.WriteLine("{0:R} = {1:R}: {2}", value1, value2,
value1.Equals(value2));
    }
}
// The example displays the following output:
//      0.3333333 = 0.33333343: False
```

Calculated values that follow different code paths and that are manipulated in different ways often prove to be unequal. In the following example, one [Single](#) value is squared, and then the square root is calculated to restore the original value. A second [Single](#) is multiplied by 3.51 and squared before the square root of the result is divided by 3.51 to restore the original value. Although the two values appear to be identical, a call to the [Equals\(Single\)](#) method indicates that they are not equal. Using the "G9" standard format string to return a result string that displays all the significant digits of each [Single](#) value shows that the second value is .000000000001 less than the first.

C#

```
using System;

public class Example1
{
    public static void Main()
    {
        float value1 = 10.201438f;
        value1 = (float)Math.Sqrt((float)Math.Pow(value1, 2));
        float value2 = (float)Math.Pow((float)value1 * 3.51f, 2);
        value2 = ((float)Math.Sqrt(value2)) / 3.51f;
        Console.WriteLine("{0} = {1}: {2}\n",
                           value1, value2, value1.Equals(value2));
        Console.WriteLine("{0:G9} = {1:G9}", value1, value2);
    }
}
// The example displays the following output:
//      10.20144 = 10.20144: False
//
//      10.201438 = 10.2014389
```

In cases where a loss of precision is likely to affect the result of a comparison, you can use the following techniques instead of calling the [Equals](#) or [CompareTo](#) method:

- Call the [Math.Round](#) method to ensure that both values have the same precision. The following example modifies a previous example to use this approach so that two fractional values are equivalent.

C#

```
using System;

public class Example2
{
    public static void Main()
    {
        float value1 = .3333333f;
        float value2 = 1.0f / 3;
        int precision = 7;
        value1 = (float)Math.Round(value1, precision);
        value2 = (float)Math.Round(value2, precision);
        Console.WriteLine("{0:R} = {1:R}: {2}", value1, value2,
value1.Equals(value2));
    }
}
// The example displays the following output:
//      0.3333333 = 0.3333333: True
```

The problem of precision still applies to rounding of midpoint values. For more information, see the [Math.Round\(Double, Int32, MidpointRounding\)](#) method.

- Test for approximate equality instead of equality. This technique requires that you define either an absolute amount by which the two values can differ but still be equal, or that you define a relative amount by which the smaller value can diverge from the larger value.

⚠ Warning

Single.Epsilon is sometimes used as an absolute measure of the distance between two **Single** values when testing for equality. However, **Single.Epsilon** measures the smallest possible value that can be added to, or subtracted from, a **Single** whose value is zero. For most positive and negative **Single** values, the value of **Single.Epsilon** is too small to be detected. Therefore, except for values that are zero, we do not recommend its use in tests for equality.

The following example uses the latter approach to define an `IsApproximatelyEqual` method that tests the relative difference between two values. It also contrasts the result of calls to the `IsApproximatelyEqual` method and the `Equals(Single)` method.

C#

```
using System;

public class Example3
{
    public static void Main()
    {
        float one1 = .1f * 10;
        float one2 = 0f;
        for (int ctr = 1; ctr <= 10; ctr++)
            one2 += .1f;

        Console.WriteLine("{0:R} = {1:R}: {2}", one1, one2,
one1.Equals(one2));
        Console.WriteLine("{0:R} is approximately equal to {1:R}: {2}",
one1, one2,
IsApproximatelyEqual(one1, one2, .000001f));
    }

    static bool IsApproximatelyEqual(float value1, float value2, float
epsilon)
    {
        // If they are equal anyway, just return True.
        if (value1.Equals(value2))
            return true;

        // Handle NaN, Infinity.
        if (Double.IsInfinity(value1) | Double.IsNaN(value1))
            return value1.Equals(value2);
        else if (Double.IsInfinity(value2) | Double.IsNaN(value2))
            return value1.Equals(value2);

        // Handle zero to avoid division by zero
        double divisor = Math.Max(value1, value2);
        if (divisor.Equals(0))
            divisor = Math.Min(value1, value2);

        return Math.Abs(value1 - value2) / divisor <= epsilon;
    }
}

// The example displays the following output:
//      1 = 1.0000012: False
//      1 is approximately equal to 1.0000012: True
```

Floating-point values and exceptions

Operations with floating-point values do not throw exceptions, unlike operations with integral types, which throw exceptions in cases of illegal operations such as division by zero or overflow. Instead, in these situations, the result of a floating-point operation is zero, positive infinity, negative infinity, or not a number (NaN):

- If the result of a floating-point operation is too small for the destination format, the result is zero. This can occur when two very small floating-point numbers are multiplied, as the following example shows.

C#

```
using System;

public class Example6
{
    public static void Main()
    {
        float value1 = 1.163287e-36f;
        float value2 = 9.164234e-25f;
        float result = value1 * value2;
        Console.WriteLine("{0} * {1} = {2}", value1, value2, result);
        Console.WriteLine("{0} = 0: {1}", result, result.Equals(0.0f));
    }
}
// The example displays the following output:
//      1.163287E-36 * 9.164234E-25 = 0
//      0 = 0: True
```

- If the magnitude of the result of a floating-point operation exceeds the range of the destination format, the result of the operation is [PositiveInfinity](#) or [NegativeInfinity](#), as appropriate for the sign of the result. The result of an operation that overflows [Single.MaxValue](#) is [PositiveInfinity](#), and the result of an operation that overflows [Single.MinValue](#) is [NegativeInfinity](#), as the following example shows.

C#

```
using System;

public class Example7
{
    public static void Main()
    {
        float value1 = 3.065e35f;
        float value2 = 6.9375e32f;
        float result = value1 * value2;
        Console.WriteLine("PositiveInfinity: {0}",
                          Single.IsPositiveInfinity(result));
        Console.WriteLine("NegativeInfinity: {0}\n",
                          Single.IsNegativeInfinity(result));
    }
}
```

```

        Single.IsNegativeInfinity(result));

        value1 = -value1;
        result = value1 * value2;
        Console.WriteLine("PositiveInfinity: {0}",
                           Single.IsPositiveInfinity(result));
        Console.WriteLine("NegativeInfinity: {0}",
                           Single.IsNegativeInfinity(result));
    }
}

// The example displays the following output:
//      PositiveInfinity: True
//      NegativeInfinity: False
//
//      PositiveInfinity: False
//      NegativeInfinity: True

```

`PositiveInfinity` also results from a division by zero with a positive dividend, and `NegativeInfinity` results from a division by zero with a negative dividend.

- If a floating-point operation is invalid, the result of the operation is `NaN`. For example, `NaN` results from the following operations:
 - Division by zero with a dividend of zero. Note that other cases of division by zero result in either `PositiveInfinity` or `NegativeInfinity`.
 - Any floating-point operation with invalid input. For example, attempting to find the square root of a negative value returns `NaN`.
 - Any operation with an argument whose value is `Single.NaN`.

Type conversions

The `Single` structure does not define any explicit or implicit conversion operators; instead, conversions are implemented by the compiler.

The following table lists the possible conversions of a value of the other primitive numeric types to a `Single` value. It also indicates whether the conversion is widening or narrowing and whether the resulting `Single` may have less precision than the original value.

[+] Expand table

| Conversion from | Widening/narrowing | Possible loss of precision |
|-------------------|--------------------|----------------------------|
| <code>Byte</code> | Widening | No |

| Conversion from | Widening/narrowing | Possible loss of precision |
|-----------------|--|--|
| Decimal | Widening Note that C# requires a cast operator. | Yes. Decimal supports 29 decimal digits of precision; Single supports 9. |
| Double | Narrowing; out-of-range values are converted to Double.NegativeInfinity or Double.PositiveInfinity . | Yes. Double supports 17 decimal digits of precision; Single supports 9. |
| Int16 | Widening | No |
| Int32 | Widening | Yes. Int32 supports 10 decimal digits of precision; Single supports 9. |
| Int64 | Widening | Yes. Int64 supports 19 decimal digits of precision; Single supports 9. |
| SByte | Widening | No |
| UInt16 | Widening | No |
| UInt32 | Widening | Yes. UInt32 supports 10 decimal digits of precision; Single supports 9. |
| UInt64 | Widening | Yes. Int64 supports 20 decimal digits of precision; Single supports 9. |

The following example converts the minimum or maximum value of other primitive numeric types to a [Single](#) value.

```

UInt32.MaxValue,
                UInt64.MinValue, UInt64.MaxValue };

    float sngValue;
    foreach (var value in values)
    {
        if (value.GetType() == typeof(Decimal) ||
            value.GetType() == typeof(Double))
            sngValue = (float)value;
        else
            sngValue = value;
        Console.WriteLine("{0} ({1}) --> {2:R} ({3})",
                            value, value.GetType().Name,
                            sngValue, sngValue.GetType().Name);
    }
}

// The example displays the following output:
//      0 (Byte) --> 0 (Single)
//      255 (Byte) --> 255 (Single)
//      -79228162514264337593543950335 (Decimal) --> -7.92281625E+28
(Single)
//      79228162514264337593543950335 (Decimal) --> 7.92281625E+28 (Single)
//      -1.79769313486232E+308 (Double) --> -Infinity (Single)
//      1.79769313486232E+308 (Double) --> Infinity (Single)
//      -32768 (Int16) --> -32768 (Single)
//      32767 (Int16) --> 32767 (Single)
//      -2147483648 (Int32) --> -2.14748365E+09 (Single)
//      2147483647 (Int32) --> 2.14748365E+09 (Single)
//      -9223372036854775808 (Int64) --> -9.223372E+18 (Single)
//      9223372036854775807 (Int64) --> 9.223372E+18 (Single)
//      -128 (SByte) --> -128 (Single)
//      127 (SByte) --> 127 (Single)
//      0 (UInt16) --> 0 (Single)
//      65535 (UInt16) --> 65535 (Single)
//      0 (UInt32) --> 0 (Single)
//      4294967295 (UInt32) --> 4.2949673E+09 (Single)
//      0 (UInt64) --> 0 (Single)
//      18446744073709551615 (UInt64) --> 1.84467441E+19 (Single)

```

In addition, the [Double](#) values [Double.NaN](#), [Double.PositiveInfinity](#), and [Double.NegativeInfinity](#) convert to [Single.NaN](#), [Single.PositiveInfinity](#), and [Single.NegativeInfinity](#), respectively.

Note that the conversion of the value of some numeric types to a [Single](#) value can involve a loss of precision. As the example illustrates, a loss of precision is possible when converting [Decimal](#), [Double](#), [Int32](#), [Int64](#), [UInt32](#), and [UInt64](#) values to [Single](#) values.

The conversion of a [Single](#) value to a [Double](#) is a widening conversion. The conversion may result in a loss of precision if the [Double](#) type does not have a precise representation for the [Single](#) value.

The conversion of a [Single](#) value to a value of any primitive numeric data type other than a [Double](#) is a narrowing conversion and requires a cast operator (in C#) or a conversion method (in Visual Basic). Values that are outside the range of the target data type, which are defined by the target type's `MinValue` and `MaxValue` properties, behave as shown in the following table.

[+] Expand table

| Target type | Result |
|-------------------------|--|
| Any integral type | An OverflowException exception if the conversion occurs in a checked context. If the conversion occurs in an unchecked context (the default in C#), the conversion operation succeeds but the value overflows. |
| Decimal | An OverflowException exception, |

In addition, [Single.NaN](#), [Single.PositiveInfinity](#), and [Single.NegativeInfinity](#) throw an [OverflowException](#) for conversions to integers in a checked context, but these values overflow when converted to integers in an unchecked context. For conversions to [Decimal](#), they always throw an [OverflowException](#). For conversions to [Double](#), they convert to [Double.NaN](#), [Double.PositiveInfinity](#), and [Double.NegativeInfinity](#), respectively.

Note that a loss of precision may result from converting a [Single](#) value to another numeric type. In the case of converting non-integral [Single](#) values, as the output from the example shows, the fractional component is lost when the [Single](#) value is either rounded (as in Visual Basic) or truncated (as in C# and F#). For conversions to [Decimal](#) values, the [Single](#) value may not have a precise representation in the target data type.

The following example converts a number of [Single](#) values to several other numeric types. The conversions occur in a checked context in Visual Basic (the default), in C# (because of the `checked` keyword), and in F# (because of the `open Checked` statement). The output from the example shows the result for conversions in both a checked and unchecked context. You can perform conversions in an unchecked context in Visual Basic by compiling with the `/removeintchecks+` compiler switch, in C# by commenting out the `checked` statement, and in F# by commenting out the `open Checked` statement.

C#

```
using System;

public class Example5
{
    public static void Main()
```

```

{
    float[] values = { Single.MinValue, -67890.1234f, -12345.6789f,
                      12345.6789f, 67890.1234f, Single.MaxValue,
                      Single.NaN, Single.PositiveInfinity,
                      Single.NegativeInfinity };
    checked
    {
        foreach (var value in values)
        {
            try
            {
                Int64 lValue = (long)value;
                Console.WriteLine("{0} ({1}) --> {2} (0x{2:X16}) ({3})",
                                  value, value.GetType().Name,
                                  lValue, lValue.GetType().Name);
            }
            catch (OverflowException)
            {
                Console.WriteLine("Unable to convert {0} to Int64.",
                                  value);
            }
            try
            {
                UInt64 ulValue = (ulong)value;
                Console.WriteLine("{0} ({1}) --> {2} (0x{2:X16}) ({3})",
                                  value, value.GetType().Name,
                                  ulValue, ulValue.GetType().Name);
            }
            catch (OverflowException)
            {
                Console.WriteLine("Unable to convert {0} to UInt64.",
                                  value);
            }
            try
            {
                Decimal dValue = (decimal)value;
                Console.WriteLine("{0} ({1}) --> {2} ({3})",
                                  value, value.GetType().Name,
                                  dValue, dValue.GetType().Name);
            }
            catch (OverflowException)
            {
                Console.WriteLine("Unable to convert {0} to Decimal.",
                                  value);
            }

            Double dblValue = value;
            Console.WriteLine("{0} ({1}) --> {2} ({3})",
                              value, value.GetType().Name,
                              dblValue, dblValue.GetType().Name);
            Console.WriteLine();
        }
    }
}

```

```
// The example displays the following output for conversions performed
// in a checked context:
//      Unable to convert -3.402823E+38 to Int64.
//      Unable to convert -3.402823E+38 to UInt64.
//      Unable to convert -3.402823E+38 to Decimal.
//      -3.402823E+38 (Single) --> -3.40282346638529E+38 (Double)
//
//      -67890.13 (Single) --> -67890 (0xFFFFFFFFFFFFF6CE) (Int64)
//      Unable to convert -67890.13 to UInt64.
//      -67890.13 (Single) --> -67890.12 (Decimal)
//      -67890.13 (Single) --> -67890.125 (Double)
//
//      -12345.68 (Single) --> -12345 (0xFFFFFFFFFFFFFCFC7) (Int64)
//      Unable to convert -12345.68 to UInt64.
//      -12345.68 (Single) --> -12345.68 (Decimal)
//      -12345.68 (Single) --> -12345.6787109375 (Double)
//
//      12345.68 (Single) --> 12345 (0x00000000000003039) (Int64)
//      12345.68 (Single) --> 12345 (0x00000000000003039) (UInt64)
//      12345.68 (Single) --> 12345.68 (Decimal)
//      12345.68 (Single) --> 12345.6787109375 (Double)
//
//      67890.13 (Single) --> 67890 (0x00000000000010932) (Int64)
//      67890.13 (Single) --> 67890 (0x00000000000010932) (UInt64)
//      67890.13 (Single) --> 67890.12 (Decimal)
//      67890.13 (Single) --> 67890.125 (Double)
//
//      Unable to convert 3.402823E+38 to Int64.
//      Unable to convert 3.402823E+38 to UInt64.
//      Unable to convert 3.402823E+38 to Decimal.
//      3.402823E+38 (Single) --> 3.40282346638529E+38 (Double)
//
//      Unable to convert NaN to Int64.
//      Unable to convert NaN to UInt64.
//      Unable to convert NaN to Decimal.
//      NaN (Single) --> NaN (Double)
//
//      Unable to convert Infinity to Int64.
//      Unable to convert Infinity to UInt64.
//      Unable to convert Infinity to Decimal.
//      Infinity (Single) --> Infinity (Double)
//
//      Unable to convert -Infinity to Int64.
//      Unable to convert -Infinity to UInt64.
//      Unable to convert -Infinity to Decimal.
//      -Infinity (Single) --> -Infinity (Double)
// The example displays the following output for conversions performed
// in an unchecked context:
//      -3.402823E+38 (Single) --> -9223372036854775808
//(0x8000000000000000) (Int64)
//      -3.402823E+38 (Single) --> 9223372036854775808 (0x8000000000000000)
(UInt64)
//      Unable to convert -3.402823E+38 to Decimal.
//      -3.402823E+38 (Single) --> -3.40282346638529E+38 (Double)
//
```

```

//      -67890.13 (Single) --> -67890 (0xFFFFFFFFFFFFFEF6CE) (Int64)
//      -67890.13 (Single) --> 18446744073709483726 (0xFFFFFFFFFFFFFEF6CE)
(UInt64)
//      -67890.13 (Single) --> -67890.12 (Decimal)
//      -67890.13 (Single) --> -67890.125 (Double)
//
//      -12345.68 (Single) --> -12345 (0xFFFFFFFFFFFFCFC7) (Int64)
//      -12345.68 (Single) --> 18446744073709539271 (0xFFFFFFFFFFFFCFC7)
(UInt64)
//      -12345.68 (Single) --> -12345.68 (Decimal)
//      -12345.68 (Single) --> -12345.6787109375 (Double)
//
//      12345.68 (Single) --> 12345 (0x0000000000003039) (Int64)
//      12345.68 (Single) --> 12345 (0x0000000000003039) (UInt64)
//      12345.68 (Single) --> 12345.68 (Decimal)
//      12345.68 (Single) --> 12345.6787109375 (Double)
//
//      67890.13 (Single) --> 67890 (0x0000000000010932) (Int64)
//      67890.13 (Single) --> 67890 (0x0000000000010932) (UInt64)
//      67890.13 (Single) --> 67890.12 (Decimal)
//      67890.13 (Single) --> 67890.125 (Double)
//
//      3.402823E+38 (Single) --> -9223372036854775808 (0x8000000000000000)
(Int64)
//      3.402823E+38 (Single) --> 0 (0x0000000000000000) (UInt64)
//      Unable to convert 3.402823E+38 to Decimal.
//      3.402823E+38 (Single) --> 3.40282346638529E+38 (Double)
//
//      NaN (Single) --> -9223372036854775808 (0x8000000000000000) (Int64)
//      NaN (Single) --> 0 (0x0000000000000000) (UInt64)
//      Unable to convert NaN to Decimal.
//      NaN (Single) --> NaN (Double)
//
//      Infinity (Single) --> -9223372036854775808 (0x8000000000000000)
(Int64)
//      Infinity (Single) --> 0 (0x0000000000000000) (UInt64)
//      Unable to convert Infinity to Decimal.
//      Infinity (Single) --> Infinity (Double)
//
//      -Infinity (Single) --> -9223372036854775808 (0x8000000000000000)
(Int64)
//      -Infinity (Single) --> 9223372036854775808 (0x8000000000000000)
(UInt64)
//      Unable to convert -Infinity to Decimal.
//      -Infinity (Single) --> -Infinity (Double)

```

For more information on the conversion of numeric types, see [Type Conversion in .NET](#) and [Type Conversion Tables](#).

Floating-point functionality

The [Single](#) structure and related types provide methods to perform the following categories of operations:

- **Comparison of values.** You can call the [Equals](#) method to determine whether two [Single](#) values are equal, or the [CompareTo](#) method to determine the relationship between two values.

The [Single](#) structure also supports a complete set of comparison operators. For example, you can test for equality or inequality, or determine whether one value is greater than or equal to another value. If one of the operands is a [Double](#), the [Single](#) value is converted to a [Double](#) before performing the comparison. If one of the operands is an integral type, it is converted to a [Single](#) before performing the comparison. Although these are widening conversions, they may involve a loss of precision.

Warning

Because of differences in precision, two [Single](#) values that you expect to be equal may turn out to be unequal, which affects the result of the comparison. See the [Test for equality](#) section for more information about comparing two [Single](#) values.

You can also call the [IsNaN](#), [IsInfinity](#), [IsPositiveInfinity](#), and [IsNegativeInfinity](#) methods to test for these special values.

- **Mathematical operations.** Common arithmetic operations such as addition, subtraction, multiplication, and division are implemented by language compilers and Common Intermediate Language (CIL) instructions rather than by [Single](#) methods. If the other operand in a mathematical operation is a [Double](#), the [Single](#) is converted to a [Double](#) before performing the operation, and the result of the operation is also a [Double](#) value. If the other operand is an integral type, it is converted to a [Single](#) before performing the operation, and the result of the operation is also a [Single](#) value.

You can perform other mathematical operations by calling `static` (Shared in Visual Basic) methods in the [System.Math](#) class. These include additional methods commonly used for arithmetic (such as [Math.Abs](#), [Math.Sign](#), and [Math.Sqrt](#)), geometry (such as [Math.Cos](#) and [Math.Sin](#)), and calculus (such as [Math.Log](#)). In all cases, the [Single](#) value is converted to a [Double](#).

You can also manipulate the individual bits in a [Single](#) value. The [BitConverter.GetBytes\(Single\)](#) method returns its bit pattern in a byte array. By

passing that byte array to the [BitConverter.ToInt32](#) method, you can also preserve the [Single](#) value's bit pattern in a 32-bit integer.

- **Rounding.** Rounding is often used as a technique for reducing the impact of differences between values caused by problems of floating-point representation and precision. You can round a [Single](#) value by calling the [Math.Round](#) method. However, note that the [Single](#) value is converted to a [Double](#) before the method is called, and the conversion can involve a loss of precision.
- **Formatting.** You can convert a [Single](#) value to its string representation by calling the [ToString](#) method or by using the [composite formatting](#) feature. For information about how format strings control the string representation of floating-point values, see the [Standard Numeric Format Strings](#) and [Custom Numeric Format Strings](#) topics.
- **Parsing strings.** You can convert the string representation of a floating-point value to a [Single](#) value by calling the [Parse](#) or [TryParse](#) method. If the parse operation fails, the [Parse](#) method throws an exception, whereas the [TryParse](#) method returns `false`.
- **Type conversion.** The [Single](#) structure provides an explicit interface implementation for the [IConvertible](#) interface, which supports conversion between any two standard .NET Framework data types. Language compilers also support the implicit conversion of values for all other standard numeric types except for the conversion of [Double](#) to [Single](#) values. Conversion of a value of any standard numeric type other than a [Double](#) to a [Single](#) is a widening conversion and does not require the use of a casting operator or conversion method.

However, conversion of 32-bit and 64-bit integer values can involve a loss of precision. The following table lists the differences in precision for 32-bit, 64-bit, and [Double](#) types:

[+] Expand table

| Type | Maximum precision (in decimal digits) | Internal precision (in decimal digits) |
|--|---------------------------------------|--|
| Double | 15 | 17 |
| Int32 and UInt32 | 10 | 10 |
| Int64 and UInt64 | 19 | 19 |

| Type | Maximum precision (in decimal digits) | Internal precision (in decimal digits) |
|--------|---------------------------------------|--|
| Single | 7 | 9 |

The problem of precision most frequently affects [Single](#) values that are converted to [Double](#) values. In the following example, two values produced by identical division operations are unequal, because one of the values is a single-precision floating point value that is converted to a [Double](#).

C#

```
using System;

public class Example8
{
    public static void Main()
    {
        Double value1 = 1 / 3.0;
        Single sValue2 = 1 / 3.0f;
        Double value2 = (Double)sValue2;
        Console.WriteLine("{0:R} = {1:R}: {2}",
                           value1, value2,
                           value1.Equals(value2));
    }
}
// The example displays the following output:
//      0.3333333333333331 = 0.3333333432674408: False
```

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

.NET is an open source project. Select a link to provide feedback:

[Open a documentation issue](#)

[Provide product feedback](#)

Dates, times, and time zones

Article • 01/03/2023

In addition to the basic [DateTime](#) structure, .NET provides the following classes that support working with time zones:

- [TimeZone](#)

Use this class to work with the system's local time zone and the Coordinated Universal Time (UTC) zone. The functionality of the [TimeZone](#) class is largely superseded by the [TimeZoneInfo](#) class.

- [TimeZoneInfo](#)

Use this class to work with any time zone that is predefined on a system, to create new time zones, and to easily convert dates and times from one time zone to another. For new development, use the [TimeZoneInfo](#) class instead of the [TimeZone](#) class.

- [DateTimeOffset](#)

Use this structure to work with dates and times whose offset (or difference) from UTC is known. The [DateTimeOffset](#) structure combines a date and time value with that time's offset from UTC. Because of its relationship to UTC, an individual date and time value unambiguously identifies a single point in time. This makes a [DateTimeOffset](#) value more portable from one computer to another than a [DateTime](#) value.

Starting with .NET 6, the following types are available:

- [DateOnly](#)

Use this structure when working with a value that only represents a date. The date represents the entire day, from the start of the day to the end. [DateOnly](#) has a range of `0001-01-01` through `9999-12-31`. And, this type represents the month, day, and year combination without a specific time. If you previously used a [DateTime](#) type in your code to represent a date that disregarded the time, use this type in its place. For more information, see [How to use the DateOnly and TimeOnly structures](#).

- [TimeOnly](#)

Use this structure to represent a time without a date. The time represents the hours, minutes, and seconds of a non-specific day. `TimeOnly` has a range of `00:00:00.0000000` to `23:59:59.9999999`. This type can be used to replace `DateTime` and `TimeSpan` types in your code when you used those types to represent a time. For more information, see [How to use the DateOnly and TimeOnly structures](#).

The next section provides the information that you need to work with time zones and to create time zone-aware applications that can convert dates and times from one time zone to another.

In this section

[Time zone overview](#)

Discusses the terminology, concepts, and issues involved in creating time zone-aware applications.

[Choosing between DateTime, DateTimeOffset, TimeSpan, and TimeZoneInfo](#)

Discusses when to use the `DateTime`, `DateTimeOffset`, and `TimeZoneInfo` types when working with date and time data.

[Finding the time zones defined on a local system](#)

Describes how to enumerate the time zones found on a local system.

[How to: Enumerate time zones present on a computer](#)

Provides examples that enumerate the time zones defined in a computer's registry and that let users select a predefined time zone from a list.

[How to: Access the predefined UTC and local time zone objects](#)

Describes how to access Coordinated Universal Time and the local time zone.

[How to: Instantiate a TimeZoneInfo object](#)

Describes how to instantiate a `TimeZoneInfo` object from the local system registry.

[Instantiating a DateTimeOffset object](#)

Discusses the ways in which a `DateTimeOffset` object can be instantiated, and the ways in which a `DateTime` value can be converted to a `DateTimeOffset` value.

[How to: Create time zones without adjustment rules](#)

Describes how to create a custom time zone that does not support the transition to and from daylight saving time.

[How to: Create time zones with adjustment rules](#)

Describes how to create a custom time zone that supports one or more transitions to

and from daylight saving time.

[Saving and restoring time zones](#)

Describes [TimeZoneInfo](#) support for serialization and deserialization of time zone data and illustrates some of the scenarios in which these features can be used.

[How to: Save time zones to an embedded resource](#)

Describes how to create a custom time zone and save its information in a resource file.

[How to: Restore time zones from an embedded resource](#)

Describes how to instantiate custom time zones that have been saved to an embedded resource file.

[Performing arithmetic operations with dates and times](#)

Discusses the issues involved in adding, subtracting, and comparing [DateTime](#) and [DateTimeOffset](#) values.

[How to: Use time zones in date and time arithmetic](#)

Discusses how to perform date and time arithmetic that reflects a time zone's adjustment rules.

[Converting between DateTime and DateTimeOffset](#)

Describes how to convert between [DateTime](#) and [DateTimeOffset](#) values.

[Converting times between time zones](#)

Describes how to convert times from one time zone to another.

[How to: Resolve ambiguous times](#)

Describes how to resolve an ambiguous time by mapping it to the time zone's standard time.

[How to: Let users resolve ambiguous times](#)

Describes how to let a user determine the mapping between an ambiguous local time and Coordinated Universal Time.

Reference

[System.TimeZoneInfo](#)



Collaborate with us on
GitHub

.NET

.NET feedback

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

Choose between DateTime, DateOnly, DateTimeOffset, TimeSpan, TimeOnly, and TimeZoneInfo

Article • 05/02/2023

.NET applications can use date and time information in several ways. The more common uses of date and time information include:

- To reflect a date only, so that time information is not important.
- To reflect a time only, so that date information is not important.
- To reflect an abstract date and time that's not tied to a specific time and place (for example, most stores in an international chain open on weekdays at 9:00 A.M.).
- To retrieve date and time information from sources outside of .NET, typically where date and time information is stored in a simple data type.
- To uniquely and unambiguously identify a single point in time. Some applications require that a date and time be unambiguous only on the host system. Other apps require that it be unambiguous across systems (that is, a date serialized on one system can be meaningfully deserialized and used on another system anywhere in the world).
- To preserve multiple related times (such as the requester's local time and the server's time of receipt for a web request).
- To perform date and time arithmetic, possibly with a result that uniquely and unambiguously identifies a single point in time.

.NET includes the [DateTime](#), [DateOnly](#), [DateTimeOffset](#), [TimeSpan](#), [TimeOnly](#), and [TimeZoneInfo](#) types, all of which can be used to build applications that work with dates and times.

ⓘ Note

This article doesn't discuss [TimeZone](#) because its functionality is almost entirely incorporated in the [TimeZoneInfo](#) class. Whenever possible, use the [TimeZoneInfo](#) class instead of the [TimeZone](#) class.

The DateTimeOffset structure

The [DateTimeOffset](#) structure represents a date and time value, together with an offset that indicates how much that value differs from UTC. Thus, the value always

unambiguously identifies a single point in time.

The [DateTimeOffset](#) type includes all of the functionality of the [DateTime](#) type along with time zone awareness. This makes it suitable for applications that:

- Uniquely and unambiguously identify a single point in time. The [DateTimeOffset](#) type can be used to unambiguously define the meaning of "now", to log transaction times, to log the times of system or application events, and to record file creation and modification times.
- Perform general date and time arithmetic.
- Preserve multiple related times, as long as those times are stored as two separate values or as two members of a structure.

ⓘ Note

These uses for [DateTimeOffset](#) values are much more common than those for [DateTime](#) values. As a result, consider [DateTimeOffset](#) as the default date and time type for application development.

A [DateTimeOffset](#) value isn't tied to a particular time zone, but can originate from a variety of time zones. The following example lists the time zones to which a number of [DateTimeOffset](#) values (including a local Pacific Standard Time) can belong.

C#

```
using System;
using System.Collections.ObjectModel;

public class TimeOffsets
{
    public static void Main()
    {
        DateTime thisDate = new DateTime(2007, 3, 10, 0, 0, 0);
        DateTime dstDate = new DateTime(2007, 6, 10, 0, 0, 0);
        DateTimeOffset thisTime;

        thisTime = new DateTimeOffset(dstDate, new TimeSpan(-7, 0, 0));
        ShowPossibleTimeZones(thisTime);

        thisTime = new DateTimeOffset(thisDate, new TimeSpan(-6, 0, 0));
        ShowPossibleTimeZones(thisTime);

        thisTime = new DateTimeOffset(thisDate, new TimeSpan(+1, 0, 0));
        ShowPossibleTimeZones(thisTime);
    }

    private static void ShowPossibleTimeZones(DateTimeOffset offsetTime)
```

```

{
    TimeSpan offset = offsetTime.Offset;
    ReadOnlyCollection<TimeZoneInfo> timeZones;

    Console.WriteLine("{0} could belong to the following time zones:",
        offsetTime.ToString());
    // Get all time zones defined on local system
    timeZones = TimeZoneInfo.GetSystemTimeZones();
    // Iterate time zones
    foreach (TimeZoneInfo timeZone in timeZones)
    {
        // Compare offset with offset for that date in that time zone
        if (timeZone.GetUtcOffset(offsetTime.DateTime).Equals(offset))
            Console.WriteLine(" {0}", timeZone.DisplayName);
    }
    Console.WriteLine();
}
// This example displays the following output to the console:
//       6/10/2007 12:00:00 AM -07:00 could belong to the following time
zones:
//           (GMT-07:00) Arizona
//           (GMT-08:00) Pacific Time (US & Canada)
//           (GMT-08:00) Tijuana, Baja California
//
//       3/10/2007 12:00:00 AM -06:00 could belong to the following time
zones:
//           (GMT-06:00) Central America
//           (GMT-06:00) Central Time (US & Canada)
//           (GMT-06:00) Guadalajara, Mexico City, Monterrey - New
//           (GMT-06:00) Guadalajara, Mexico City, Monterrey - Old
//           (GMT-06:00) Saskatchewan
//
//       3/10/2007 12:00:00 AM +01:00 could belong to the following time
zones:
//           (GMT+01:00) Amsterdam, Berlin, Bern, Rome, Stockholm, Vienna
//           (GMT+01:00) Belgrade, Bratislava, Budapest, Ljubljana, Prague
//           (GMT+01:00) Brussels, Copenhagen, Madrid, Paris
//           (GMT+01:00) Sarajevo, Skopje, Warsaw, Zagreb
//           (GMT+01:00) West Central Africa

```

The output shows that each date and time value in this example can belong to at least three different time zones. The [DateTimeOffset](#) value of 6/10/2007 shows that if a date and time value represents a daylight saving time, its offset from UTC doesn't even necessarily correspond to the originating time zone's base UTC offset or to the offset from UTC found in its display name. Because a single [DateTimeOffset](#) value isn't tightly coupled with its time zone, it can't reflect a time zone's transition to and from daylight saving time. This can be problematic when date and time arithmetic is used to manipulate a [DateTimeOffset](#) value. For a discussion of how to perform date and time

arithmetic in a way that takes account of a time zone's adjustment rules, see [Performing arithmetic operations with dates and times](#).

The DateTime structure

A [DateTime](#) value defines a particular date and time. It includes a [Kind](#) property that provides limited information about the time zone to which that date and time belongs. The [DateTimeKind](#) value returned by the [Kind](#) property indicates whether the [DateTime](#) value represents the local time ([DateTimeKind.Local](#)), Coordinated Universal Time (UTC) ([DateTimeKind.Utc](#)), or an unspecified time ([DateTimeKind.Unspecified](#)).

The [DateTime](#) structure is suitable for applications with one or more of the following characteristics:

- Work with abstract dates and times.
- Work with dates and times for which time zone information is missing.
- Work with UTC dates and times only.
- Perform date and time arithmetic, but are concerned with general results. For example, in an addition operation that adds six months to a particular date and time, it is often not important whether the result is adjusted for daylight saving time.

Unless a particular [DateTime](#) value represents UTC, that date and time value is often ambiguous or limited in its portability. For example, if a [DateTime](#) value represents the local time, it's portable within that local time zone (that is, if the value is serialized on another system in the same time zone, that value still unambiguously identifies a single point in time). Outside the local time zone, that [DateTime](#) value can have multiple interpretations. If the value's [Kind](#) property is [DateTimeKind.Unspecified](#), it's even less portable: it is now ambiguous within the same time zone and possibly even on the same system where it was first serialized. Only if a [DateTime](#) value represents UTC does that value unambiguously identify a single point in time regardless of the system or time zone in which the value is used.

 **Important**

When saving or sharing [DateTime](#) data, use UTC and set the [DateTime](#) value's [Kind](#) property to [DateTimeKind.Utc](#).

The DateOnly structure

The `DateOnly` structure represents a specific date, without time. Since it has no time component, it represents a date from the start of the day to the end of the day. This structure is ideal for storing specific dates, such as a birth date, an anniversary date, a holiday, or a business-related date.

Although you could use `DateTime` while ignoring the time component, there are a few benefits to using `DateOnly` over `DateTime`:

- The `DateTime` structure may roll into the previous or next day if it's offset by a time zone. `DateOnly` can't be offset by a time zone, and it always represents the date that was set.
- Serializing a `DateTime` structure includes the time component, which may obscure the intent of the data. Also, `DateOnly` serializes less data.
- When code interacts with a database, such as SQL Server, whole dates are generally stored as the `date` data type, which doesn't include a time. `DateOnly` matches the database type better.

For more information about `DateOnly`, see [How to use the DateOnly and TimeOnly structures](#).

ⓘ Important

`DateOnly` isn't available in .NET Framework.

The `TimeSpan` structure

The `TimeSpan` structure represents a time interval. Its two typical uses are:

- Reflecting the time interval between two date and time values. For example, subtracting one `DateTime` value from another returns a `TimeSpan` value.
- Measuring elapsed time. For example, the `Stopwatch.Elapsed` property returns a `TimeSpan` value that reflects the time interval that has elapsed since the call to one of the `Stopwatch` methods that begins to measure elapsed time.

A `TimeSpan` value can also be used as a replacement for a `DateTime` value when that value reflects a time without reference to a particular day. This usage is similar to the `DateTime.TimeOfDay` and `DateTimeOffset.TimeOfDay` properties, which return a `TimeSpan` value that represents the time without reference to a date. For example, the `TimeSpan` structure can be used to reflect a store's daily opening or closing time, or it can be used to represent the time at which any regular event occurs.

The following example defines a `StoreInfo` structure that includes `TimeSpan` objects for store opening and closing times, as well as a `TimeZoneInfo` object that represents the store's time zone. The structure also includes two methods, `IsOpenNow` and `IsOpenAt`, that indicates whether the store is open at a time specified by the user, who is assumed to be in the local time zone.

C#

```
using System;

public struct StoreInfo
{
    public String store;
    public TimeZoneInfo tz;
    public TimeSpan open;
    public TimeSpan close;

    public bool IsOpenNow()
    {
        return IsOpenAt(DateTime.Now.TimeOfDay);
    }

    public bool IsOpenAt(TimeSpan time)
    {
        TimeZoneInfo local = TimeZoneInfo.Local;
        TimeSpan offset = TimeZoneInfo.Local.BaseUtcOffset;

        // Is the store in the same time zone?
        if (tz.Equals(local)) {
            return time >= open & time <= close;
        }
        else {
            TimeSpan delta = TimeSpan.Zero;
            TimeSpan storeDelta = TimeSpan.Zero;

            // Is it daylight saving time in either time zone?
            if (local.IsDaylightSavingTime(DateTime.Now.Date + time))
                delta = local.GetAdjustmentRules()
[local.GetAdjustmentRules().Length - 1].DaylightDelta;

            if
(tz.IsDaylightSavingTime(TimeZoneInfo.ConvertTime(DateTime.Now.Date + time,
local, tz)))
                storeDelta = tz.GetAdjustmentRules()
[tz.GetAdjustmentRules().Length - 1].DaylightDelta;

            TimeSpan comparisonTime = time + (offset -
tz.BaseUtcOffset).Negate() + (delta - storeDelta).Negate();
            return comparisonTime >= open & comparisonTime <= close;
        }
    }
}
```

```
    }  
}
```

The `StoreInfo` structure can then be used by client code like the following.

C#

```
public class Example  
{  
    public static void Main()  
    {  
        // Instantiate a StoreInfo object.  
        var store103 = new StoreInfo();  
        store103.store = "Store #103";  
        store103.tz = TimeZoneInfo.FindSystemTimeZoneById("Eastern Standard  
Time");  
        // Store opens at 8:00.  
        store103.open = new TimeSpan(8, 0, 0);  
        // Store closes at 9:30.  
        store103.close = new TimeSpan(21, 30, 0);  
  
        Console.WriteLine("Store is open now at {0}: {1}",  
                          DateTime.Now.TimeOfDay, store103.IsOpenNow());  
        TimeSpan[] times = { new TimeSpan(8, 0, 0), new TimeSpan(21, 0, 0),  
                            new TimeSpan(4, 59, 0), new TimeSpan(18, 31, 0)  
};  
        foreach (var time in times)  
        {  
            Console.WriteLine("Store is open at {0}: {1}",  
                             time, store103.IsOpenAt(time));  
        }  
    }  
    // The example displays the following output:  
    //      Store is open now at 15:29:01.6129911: True  
    //      Store is open at 08:00:00: True  
    //      Store is open at 21:00:00: False  
    //      Store is open at 04:59:00: False  
    //      Store is open at 18:31:00: False
```

The TimeOnly structure

The `TimeOnly` structure represents a time-of-day value, such as a daily alarm clock or what time you eat lunch each day. `TimeOnly` is limited to the range of `00:00:00.0000000` - `23:59:59.9999999`, a specific time of day.

Prior to the `TimeOnly` type being introduced, programmers typically used either the `DateTime` type or the `TimeSpan` type to represent a specific time. However, using these structures to simulate a time without a date may introduce some problems, which `TimeOnly` solves:

- `TimeSpan` represents elapsed time, such as time measured with a stopwatch. The upper range is more than 29,000 years, and its value can be negative to indicate moving backwards in time. A negative `TimeSpan` doesn't indicate a specific time of the day.
- If `TimeSpan` is used as a time of day, there's a risk that it could be manipulated to a value outside of the 24-hour day. `TimeOnly` doesn't have this risk. For example, if an employee's work shift starts at 18:00 and lasts for 8 hours, adding 8 hours to the `TimeOnly` structure rolls over to 2:00.
- Using `DateTime` for a time of day requires that an arbitrary date be associated with the time, and then later disregarded. It's common practice to choose `DateTime.MinValue` (0001-01-01) as the date, however, if hours are subtracted from the `DateTime` value, an `OutOfRange` exception might occur. `TimeOnly` doesn't have this problem as the time rolls forwards and backwards around the 24-hour timeframe.
- Serializing a `DateTime` structure includes the date component, which may obscure the intent of the data. Also, `TimeOnly` serializes less data.

For more information about `TimeOnly`, see [How to use the DateOnly and TimeOnly structures](#).

Important

`TimeOnly` isn't available in .NET Framework.

The `TimeZoneInfo` class

The `TimeZoneInfo` class represents any of the Earth's time zones, and enables the conversion of any date and time in one time zone to its equivalent in another time zone. The `TimeZoneInfo` class makes it possible to work with dates and times so that any date and time value unambiguously identifies a single point in time. The `TimeZoneInfo` class is also extensible. Although it depends on time zone information provided for Windows systems and defined in the registry, it supports the creation of custom time zones. It also supports the serialization and deserialization of time zone information.

In some cases, taking full advantage of the `TimeZoneInfo` class may require further development work. If date and time values are not tightly coupled with the time zones to which they belong, further work is required. Unless your application provides some mechanism for linking a date and time with its associated time zone, it's easy for a particular date and time value to become disassociated from its time zone. One method

of linking this information is to define a class or structure that contains both the date and time value and its associated time zone object.

To take advantage of time zone support in .NET, you must know the time zone to which a date and time value belongs when that date and time object is instantiated. The time zone is often not known, particularly in web or network apps.

See also

- [Dates, times, and time zones](#)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

Work with calendars

Article • 08/27/2022

Although a date and time value represents a moment in time, its string representation is culture-sensitive and depends both on the conventions used for displaying date and time values by a specific culture and on the calendar used by that culture. This topic explores the support for calendars in .NET and discusses the use of the calendar classes when working with date values.

Calendars in .NET

All calendars in .NET derive from the [System.Globalization.Calendar](#) class, which provides the base calendar implementation. One of the classes that inherits from the [Calendar](#) class is the [EastAsianLunisolarCalendar](#) class, which is the base class for all lunisolar calendars. .NET includes the following calendar implementations:

- [ChineseLunisolarCalendar](#), which represents the Chinese lunisolar calendar.
- [GregorianCalendar](#), which represents the Gregorian calendar. This calendar is further divided into subtypes (such as Arabic and Middle East French) that are defined by the [System.Globalization.GregorianCalendarTypes](#) enumeration. The [GregorianCalendar.CalendarType](#) property specifies the subtype of the Gregorian calendar.
- [HebrewCalendar](#), which represents the Hebrew calendar.
- [HijriCalendar](#), which represents the Hijri calendar.
- [JapaneseCalendar](#), which represents the Japanese calendar.
- [JapaneseLunisolarCalendar](#), which represents the Japanese lunisolar calendar.
- [JulianCalendar](#), which represents the Julian calendar.
- [KoreanCalendar](#), which represents the Korean calendar.
- [KoreanLunisolarCalendar](#), which represents the Korean lunisolar calendar.
- [PersianCalendar](#), which represents the Persian calendar.
- [TaiwanCalendar](#), which represents the Taiwan calendar.
- [TaiwanLunisolarCalendar](#), which represents the Taiwan lunisolar calendar.

- [ThaiBuddhistCalendar](#), which represents the Thai Buddhist calendar.
- [UmAlQuraCalendar](#), which represents the Um Al Qura calendar.

A calendar can be used in one of two ways:

- As the calendar used by a specific culture. Each [CultureInfo](#) object has a current calendar, which is the calendar that the object is currently using. The string representations of all date and time values automatically reflect the current culture and its current calendar. Typically, the current calendar is the culture's default calendar. [CultureInfo](#) objects also have optional calendars, which include additional calendars that the culture can use.
- As a standalone calendar independent of a specific culture. In this case, [Calendar](#) methods are used to express dates as values that reflect the calendar.

Note that six calendar classes – [ChineseLunisolarCalendar](#), [JapaneseLunisolarCalendar](#), [JulianCalendar](#), [KoreanLunisolarCalendar](#), [PersianCalendar](#), and [TaiwanLunisolarCalendar](#) – can be used only as standalone calendars. They are not used by any culture as either the default calendar or as an optional calendar.

Calendars and cultures

Each culture has a default calendar, which is defined by the [CultureInfo.Calendar](#) property. The [CultureInfo.OptionalCalendars](#) property returns an array of [Calendar](#) objects that specifies all the calendars supported by a particular culture, including that culture's default calendar.

The following example illustrates the [CultureInfo.Calendar](#) and [CultureInfo.OptionalCalendars](#) properties. It creates [CultureInfo](#) objects for the Thai (Thailand) and Japanese (Japan) cultures and displays their default and optional calendars. Note that in both cases, the culture's default calendar is also included in the [CultureInfo.OptionalCalendars](#) collection.

C#

```
using System;
using System.Globalization;

public class Example
{
    public static void Main()
    {
        // Create a CultureInfo for Thai in Thailand.
        CultureInfo th = CultureInfo.CreateSpecificCulture("th-TH");
    }
}
```

```
        DisplayCalendars(th);

        // Create a CultureInfo for Japanese in Japan.
        CultureInfo ja = CultureInfo.CreateSpecificCulture("ja-JP");
        DisplayCalendars(ja);
    }

    static void DisplayCalendars(CultureInfo ci)
    {
        Console.WriteLine("Calendars for the {0} culture:", ci.Name);

        // Get the culture's default calendar.
        Calendar defaultCalendar = ci.Calendar;
        Console.Write("    Default Calendar: {0}",
            GetCalendarName(defaultCalendar));

        if (defaultCalendar is GregorianCalendar)
            Console.WriteLine(" ({0})",
                ((GregorianCalendar)
defaultCalendar).CalendarType);
        else
            Console.WriteLine();

        // Get the culture's optional calendars.
        Console.WriteLine("    Optional Calendars:");
        foreach (var optionalCalendar in ci.OptionalCalendars) {
            Console.Write("{0,6}{1}", "", GetCalendarName(optionalCalendar));
            if (optionalCalendar is GregorianCalendar)
                Console.WriteLine(" ({0})",
                    ((GregorianCalendar)
optionalCalendar).CalendarType);

            Console.WriteLine();
        }
        Console.WriteLine();
    }

    static string GetCalendarName(Calendar cal)
    {
        return cal.ToString().Replace("System.Globalization.", "");
    }
}

// The example displays the following output:
//     Calendars for the th-TH culture:
//         Default Calendar: ThaiBuddhistCalendar
//         Optional Calendars:
//             ThaiBuddhistCalendar
//             GregorianCalendar (Localized)
//
//     Calendars for the ja-JP culture:
//         Default Calendar: GregorianCalendar (Localized)
//         Optional Calendars:
//             GregorianCalendar (Localized)
//             JapaneseCalendar
//             GregorianCalendar (USEnglish)
```

The calendar currently in use by a particular `CultureInfo` object is defined by the culture's `DateFormatInfo.Calendar` property. A culture's `DateFormatInfo` object is returned by the `CultureInfo.DateTimeFormat` property. When a culture is created, its default value is the same as the value of the `CultureInfo.Calendar` property. However, you can change the culture's current calendar to any calendar contained in the array returned by the `CultureInfo.OptionalCalendars` property. If you try to set the current calendar to a calendar that is not included in the `CultureInfo.OptionalCalendars` property value, an `ArgumentException` is thrown.

The following example changes the calendar used by the Arabic (Saudi Arabia) culture. It first instantiates a `DateTime` value and displays it using the current culture - which, in this case, is English (United States) - and the current culture's calendar (which, in this case, is the Gregorian calendar). Next, it changes the current culture to Arabic (Saudi Arabia) and displays the date using its default Um Al-Qura calendar. It then calls the `CalendarExists` method to determine whether the Hijri calendar is supported by the Arabic (Saudi Arabia) culture. Because the calendar is supported, it changes the current calendar to Hijri and again displays the date. Note that in each case, the date is displayed using the current culture's current calendar.

C#

```
using System;
using System.Globalization;
using System.Threading;

public class Example
{
    public static void Main()
    {
        DateTime date1 = new DateTime(2011, 6, 20);

        DisplayCurrentInfo();
        // Display the date using the current culture and calendar.
        Console.WriteLine(date1.ToString("d"));
        Console.WriteLine();

        CultureInfo arSA = CultureInfo.CreateSpecificCulture("ar-SA");

        // Change the current culture to Arabic (Saudi Arabia).
        Thread.CurrentThread.CurrentCulture = arSA;
        // Display date and information about the current culture.
        DisplayCurrentInfo();
        Console.WriteLine(date1.ToString("d"));
        Console.WriteLine();

        // Change the calendar to Hijri.
        Calendar hijri = new HijriCalendar();
```

```

        if (CalendarExists(arSA, hijri)) {
            arSA.DateTimeFormat.Calendar = hijri;
            // Display date and information about the current culture.
            DisplayCurrentInfo();
            Console.WriteLine(date1.ToString("d"));
        }
    }

    private static void DisplayCurrentInfo()
    {
        Console.WriteLine("Current Culture: {0}",
                          CultureInfo.CurrentCulture.Name);
        Console.WriteLine("Current Calendar: {0}",
                          DateTimeFormatInfo.CurrentInfo.Calendar);
    }

    private static bool CalendarExists(CultureInfo culture, Calendar cal)
    {
        foreach (Calendar optionalCalendar in culture.OptionalCalendars)
            if (cal.ToString().Equals(optionalCalendar.ToString()))
                return true;

        return false;
    }
}

// The example displays the following output:
//   Current Culture: en-US
//   Current Calendar: System.Globalization.GregorianCalendar
//   6/20/2011
//
//   Current Culture: ar-SA
//   Current Calendar: System.Globalization.UmAlQuraCalendar
//   18/07/32
//
//   Current Culture: ar-SA
//   Current Calendar: System.Globalization.HijriCalendar
//   19/07/32

```

Dates and calendars

With the exception of the constructors that include a parameter of type [Calendar](#) and allow the elements of a date (that is, the month, the day, and the year) to reflect values in a designated calendar, both [DateTime](#) and [DateTimeOffset](#) values are always based on the Gregorian calendar. This means, for example, that the [DateTime.Year](#) property returns the year in the Gregorian calendar, and the [DateTime.Day](#) property returns the day of the month in the Gregorian calendar.

 **Important**

It is important to remember that there is a difference between a date value and its string representation. The former is based on the Gregorian calendar; the latter is based on the current calendar of a specific culture.

The following example illustrates this difference between `DateTime` properties and their corresponding `Calendar` methods. In the example, the current culture is Arabic (Egypt), and the current calendar is Um Al Qura. A `DateTime` value is set to the fifteenth day of the seventh month of 2011. It is clear that this is interpreted as a Gregorian date, because these same values are returned by the `DateTime.ToString(String, IFormatProvider)` method when it uses the conventions of the invariant culture. The string representation of the date that is formatted using the conventions of the current culture is 14/08/32, which is the equivalent date in the Um Al Qura calendar. Next, members of `DateTime` and `Calendar` are used to return the day, the month, and the year of the `DateTime` value. In each case, the values returned by `DateTime` members reflect values in the Gregorian calendar, whereas values returned by `UmAlQuraCalendar` members reflect values in the Umm al-Qura calendar.

C#

```
using System;
using System.Globalization;
using System.Threading;

public class Example
{
    public static void Main()
    {
        // Make Arabic (Egypt) the current culture
        // and Umm al-Qura calendar the current calendar.
        CultureInfo arEG = CultureInfo.CreateSpecificCulture("ar-EG");
        Calendar cal = new UmAlQuraCalendar();
        arEG.DateTimeFormat.Calendar = cal;
        Thread.CurrentThread.CurrentCulture = arEG;

        // Display information on current culture and calendar.
        DisplayCurrentInfo();

        // Instantiate a date object.
        DateTime date1 = new DateTime(2011, 7, 15);

        // Display the string representation of the date.
        Console.WriteLine("Date: {0:d}", date1);
        Console.WriteLine("Date in the Invariant Culture: {0}",
            date1.ToString("d", CultureInfo.InvariantCulture));
        Console.WriteLine();

        // Compare DateTime properties and Calendar methods.
        Console.WriteLine("DateTime.Month property: {0}", date1.Month);
```

```

        Console.WriteLine("UmAlQura.GetMonth: {0}",
                           cal.GetMonth(date1));
        Console.WriteLine();

        Console.WriteLine("DateTime.Day property: {0}", date1.Day);
        Console.WriteLine("UmAlQura.GetDayOfMonth: {0}",
                           cal.GetDayOfMonth(date1));
        Console.WriteLine();

        Console.WriteLine("DateTime.Year property: {0:D4}", date1.Year);
        Console.WriteLine("UmAlQura.GetYear: {0}",
                           cal.GetYear(date1));
        Console.WriteLine();
    }

    private static void DisplayCurrentInfo()
    {
        Console.WriteLine("Current Culture: {0}",
                           CultureInfo.CurrentCulture.Name);
        Console.WriteLine("Current Calendar: {0}",
                           DateTimeFormatInfo.CurrentInfo.Calendar);
    }
}

// The example displays the following output:
//   Current Culture: ar-EG
//   Current Calendar: System.Globalization.UmAlQuraCalendar
//   Date: 14/08/32
//   Date in the Invariant Culture: 07/15/2011
//
//   DateTime.Month property: 7
//   UmAlQura.GetMonth: 8
//
//   DateTime.Day property: 15
//   UmAlQura.GetDayOfMonth: 14
//
//   DateTime.Year property: 2011
//   UmAlQura.GetYear: 1432

```

Instantiate dates based on a calendar

Because [DateTime](#) and [DateTimeOffset](#) values are based on the Gregorian calendar, you must call an overloaded constructor that includes a parameter of type [Calendar](#) to instantiate a date value if you want to use the day, month, or year values from a different calendar. You can also call one of the overloads of a specific calendar's [Calendar.ToDateTime](#) method to instantiate a [DateTime](#) object based on the values of a particular calendar.

The following example instantiates one [DateTime](#) value by passing a [HebrewCalendar](#) object to a [DateTime](#) constructor, and instantiates a second [DateTime](#) value by calling the [HebrewCalendar.ToDateTime\(Int32, Int32, Int32, Int32, Int32, Int32, Int32\)](#)

method. Because the two values are created with identical values from the Hebrew calendar, the call to the `DateTime.Equals` method shows that the two `DateTime` values are equal.

```
C#
```

```
using System;
using System.Globalization;

public class Example
{
    public static void Main()
    {
        HebrewCalendar hc = new HebrewCalendar();

        DateTime date1 = new DateTime(5771, 6, 1, hc);
        DateTime date2 = hc.ToDateTime(5771, 6, 1, 0, 0, 0);

        Console.WriteLine("{0:d} (Gregorian) = {1:d2}/{2:d2}/{3:d4} ({4}):{5}",
            date1,
            hc.GetMonth(date2),
            hc.GetDayOfMonth(date2),
            hc.GetYear(date2),
            GetCalendarName(hc),
            date1.Equals(date2));
    }

    private static string GetCalendarName(Calendar cal)
    {
        return cal.ToString().Replace("System.Globalization.", "").Replace("Calendar", "");
    }
}

// The example displays the following output:
//      2/5/2011 (Gregorian) = 06/01/5771 (Hebrew): True
```

Represent dates in the current calendar

Date and time formatting methods always use the current calendar when converting dates to strings. This means that the string representation of the year, the month, and the day of the month reflect the current calendar, and do not necessarily reflect the Gregorian calendar.

The following example shows how the current calendar affects the string representation of a date. It changes the current culture to Chinese (Traditional, Taiwan), and instantiates a date value. It then displays the current calendar and the date, changes the current calendar to `TaiwanCalendar`, and displays the current calendar and date once again. The

first time the date is displayed, it is represented as a date in the Gregorian calendar. The second time it is displayed, it is represented as a date in the Taiwan calendar.

C#

```
using System;
using System.Globalization;
using System.Threading;

public class Example
{
    public static void Main()
    {
        // Change the current culture to zh-TW.
        CultureInfo zhTW = CultureInfo.CreateSpecificCulture("zh-TW");
        Thread.CurrentThread.CurrentCulture = zhTW;
        // Define a date.
        DateTime date1 = new DateTime(2011, 1, 16);

        // Display the date using the default (Gregorian) calendar.
        Console.WriteLine("Current calendar: {0}",
                          zhTW.DateTimeFormat.Calendar);
        Console.WriteLine(date1.ToString("d"));

        // Change the current calendar and display the date.
        zhTW.DateTimeFormat.Calendar = new TaiwanCalendar();
        Console.WriteLine("Current calendar: {0}",
                          zhTW.DateTimeFormat.Calendar);
        Console.WriteLine(date1.ToString("d"));
    }
}

// The example displays the following output:
//   Current calendar: System.Globalization.GregorianCalendar
//   2011/1/16
//   Current calendar: System.Globalization.TaiwanCalendar
//   100/1/16
```

Represent dates in a non-current calendar

To represent a date using a calendar that is not the current calendar of a particular culture, you must call methods of that [Calendar](#) object. For example, the [Calendar.GetYear](#), [Calendar.GetMonth](#), and [Calendar.GetDayOfMonth](#) methods convert the year, month, and day to values that reflect a particular calendar.

Warning

Because some calendars are not optional calendars of any culture, representing dates in these calendars always requires that you call calendar methods. This is true

of all calendars that derive from the [EastAsianLunisolarCalendar](#), [JulianCalendar](#), and [PersianCalendar](#) classes.

The following example uses a [JulianCalendar](#) object to instantiate a date, January 9, 1905, in the Julian calendar. When this date is displayed using the default (Gregorian) calendar, it is represented as January 22, 1905. Calls to individual [JulianCalendar](#) methods enable the date to be represented in the Julian calendar.

C#

```
using System;
using System.Globalization;

public class Example
{
    public static void Main()
    {
        JulianCalendar julian = new JulianCalendar();
        DateTime date1 = new DateTime(1905, 1, 9, julian);

        Console.WriteLine("Date ({0}): {1:d}",
            CultureInfo.CurrentCulture.Calendar,
            date1);
        Console.WriteLine("Date in Julian calendar: {0:d2}/{1:d2}/{2:d4}",
            julian.GetMonth(date1),
            julian.GetDayOfMonth(date1),
            julian.GetYear(date1));
    }
}

// The example displays the following output:
//      Date (System.Globalization.GregorianCalendar): 1/22/1905
//      Date in Julian calendar: 01/09/1905
```

Calendars and date ranges

The earliest date supported by a calendar is indicated by that calendar's [Calendar.MinSupportedDateTime](#) property. For the [GregorianCalendar](#) class, that date is January 1, 0001 C.E. Most of the other calendars in .NET support a later date. Trying to work with a date and time value that precedes a calendar's earliest supported date throws an [ArgumentOutOfRangeException](#) exception.

However, there is one important exception. The default (uninitialized) value of a [DateTime](#) object and a [DateTimeOffset](#) object is equal to the [GregorianCalendar.MinSupportedDateTime](#) value. If you try to format this date in a calendar that does not support January 1, 0001 C.E. and you do not provide a format specifier, the formatting method uses the "s" (sortable date/time pattern) format

specifier instead of the "G" (general date/time pattern) format specifier. As a result, the formatting operation does not throw an [ArgumentOutOfRangeException](#) exception. Instead, it returns the unsupported date. This is illustrated in the following example, which displays the value of [DateTime.MinValue](#) when the current culture is set to Japanese (Japan) with the Japanese calendar, and to Arabic (Egypt) with the Um Al Qura calendar. It also sets the current culture to English (United States) and calls the [DateTime.ToString\(IFormatProvider\)](#) method with each of these [CultureInfo](#) objects. In each case, the date is displayed by using the sortable date/time pattern.

C#

```
using System;
using System.Globalization;
using System.Threading;

public class Example
{
    public static void Main()
    {
        DateTime dat = DateTime.MinValue;

        // Change the current culture to ja-JP with the Japanese Calendar.
        CultureInfo jaJP = CultureInfo.CreateSpecificCulture("ja-JP");
        jaJP.DateTimeFormat.Calendar = new JapaneseCalendar();
        Thread.CurrentThread.CurrentCulture = jaJP;
        Console.WriteLine("Earliest supported date by {1} calendar: {0:d}",
                          jaJP.DateTimeFormat.Calendar.MinSupportedDateTime,
                          GetCalendarName(jaJP));
        // Attempt to display the date.
        Console.WriteLine(dat.ToString());
        Console.WriteLine();

        // Change the current culture to ar-EG with the Um Al Qura calendar.
        CultureInfo arEG = CultureInfo.CreateSpecificCulture("ar-EG");
        arEG.DateTimeFormat.Calendar = new UmAlQuraCalendar();
        Thread.CurrentThread.CurrentCulture = arEG;
        Console.WriteLine("Earliest supported date by {1} calendar: {0:d}",
                          arEG.DateTimeFormat.Calendar.MinSupportedDateTime,
                          GetCalendarName(arEG));
        // Attempt to display the date.
        Console.WriteLine(dat.ToString());
        Console.WriteLine();

        // Change the current culture to en-US.
        Thread.CurrentThread.CurrentCulture =
        CultureInfo.CreateSpecificCulture("en-US");
        Console.WriteLine(dat.ToString(jaJP));
        Console.WriteLine(dat.ToString(arEG));
        Console.WriteLine(dat.ToString("d"));
    }

    private static string GetCalendarName(CultureInfo culture)
```

```

    {
        Calendar cal = culture.DateTimeFormat.Calendar;
        return cal.GetType().Name.Replace("System.Globalization.",
        "").Replace("Calendar", "");
    }
}

// The example displays the following output:
//      Earliest supported date by Japanese calendar: 明治 1/9/8
//      0001-01-01T00:00:00
//
//      Earliest supported date by UmAlQura calendar: 01/01/18
//      0001-01-01T00:00:00
//
//      0001-01-01T00:00:00
//      0001-01-01T00:00:00
//      1/1/0001

```

Work with eras

Calendars typically divide dates into eras. However, the [Calendar](#) classes in .NET do not support every era defined by a calendar, and most of the [Calendar](#) classes support only a single era. Only the [JapaneseCalendar](#) and [JapaneseLunisolarCalendar](#) classes support multiple eras.

Important

The Reiwa era, a new era in the [JapaneseCalendar](#) and [JapaneseLunisolarCalendar](#), begins on May 1, 2019. This change affects all applications that use these calendars. See the following articles for more information:

- [Handling a new era in the Japanese calendar in .NET](#), which documents features added to .NET to support calendars with multiple eras and discusses best practices to use when handling multi-era calendars.
- [Prepare your application for the Japanese era change](#), which provides information on testing your applications on Windows to ensure their readiness for the era change.
- [Summary of new Japanese Era updates for .NET Framework](#), which lists .NET Framework updates for individual Windows versions that are related to the new Japanese calendar era, notes new .NET Framework features for multi-era support, and includes things to look for in testing your applications.

An era in most calendars denotes an extremely long time period. In the Gregorian calendar, for example, the current era spans more than two millennia. For the [JapaneseCalendar](#) and the [JapaneseLunisolarCalendar](#), the two calendars that support multiple eras, this is not the case. An era corresponds to the period of an emperor's reign. Support for multiple eras, particularly when the upper limit of the current era is unknown, poses special challenges.

Eras and era names

In .NET, integers that represent the eras supported by a particular calendar implementation are stored in reverse order in the [Calendar.Eras](#) array. The current era (which is the era with the latest time range) is at index zero, and for [Calendar](#) classes that support multiple eras, each successive index reflects the previous era. The static [Calendar.CurrentEra](#) property defines the index of the current era in the [Calendar.Eras](#) array; it is a constant whose value is always zero. Individual [Calendar](#) classes also include static fields that return the value of the current era. They are listed in the following table.

| Calendar class | Current era field |
|---|---------------------------------|
| ChineseLunisolarCalendar | ChineseEra |
| GregorianCalendar | ADEra |
| HebrewCalendar | HebrewEra |
| HijriCalendar | HijriEra |
| JapaneseLunisolarCalendar | JapaneseEra |
| JulianCalendar | JulianEra |
| KoreanCalendar | KoreanEra |
| KoreanLunisolarCalendar | GregorianEra |
| PersianCalendar | PersianEra |
| ThaiBuddhistCalendar | ThaiBuddhistEra |
| UmAlQuraCalendar | UmAlQuraEra |

The name that corresponds to a particular era number can be retrieved by passing the era number to the [DateTimeFormatInfo.GetEraName](#) or [DateTimeFormatInfo.GetAbbreviatedEraName](#) method. The following example calls these methods to retrieve information about era support in the [GregorianCalendar](#) class. It displays the Gregorian calendar date that corresponds to January 1 of the second year

of the current era, as well as the Gregorian calendar date that corresponds to January 1 of the second year of each supported Japanese calendar era.

```
C#
```

```
using System;
using System.Globalization;

public class Example
{
    public static void Main()
    {
        int year = 2;
        int month = 1;
        int day = 1;
        Calendar cal = new JapaneseCalendar();

        Console.WriteLine("\nDate instantiated without an era:");
        DateTime date1 = new DateTime(year, month, day, 0, 0, 0, cal);
        Console.WriteLine("{0}/{1}/{2} in Japanese Calendar -> {3:d} in
Gregorian",
                          cal.GetMonth(date1), cal.GetDayOfMonth(date1),
                          cal.GetYear(date1), date1);

        Console.WriteLine("\nDates instantiated with eras:");
        foreach (int era in cal.Eras) {
            DateTime date2 = cal.ToDateTime(year, month, day, 0, 0, 0, 0, era);
            Console.WriteLine("{0}/{1}/{2} era {3} in Japanese Calendar ->
{4:d} in Gregorian",
                          cal.GetMonth(date2), cal.GetDayOfMonth(date2),
                          cal.GetYear(date2), cal.GetEra(date2), date2);
        }
    }
}
```

In addition, the "g" custom date and time format string includes a calendar's era name in the string representation of a date and time. For more information, see [Custom date and time format strings](#).

Instantiate a date with an era

For the two [Calendar](#) classes that support multiple eras, a date that consists of a particular year, month, and day of the month value can be ambiguous. For example, all eras supported by the [JapaneseCalendar](#) have years whose number is 1. Ordinarily, if an era is not specified, both date and time and calendar methods assume that values belong to the current era. This is true of the [DateTime](#) and [DateTimeOffset](#) constructors that include parameters of type [Calendar](#), as well as the [JapaneseCalendar.ToDateTime](#) and [JapaneseLunisolarCalendar.ToDateTime](#) methods. The following example

instantiates a date that represents January 1 of the second year of an unspecified era. If you execute the example when the Reiwa era is the current era, the date is interpreted as the second year of the Reiwa era. The era, 令和, precedes the year in the string returned by the [DateTime.ToString\(String, IFormatProvider\)](#) method and corresponds to January 1, 2020, in the Gregorian calendar. (The Reiwa era begins in the year 2019 of the Gregorian calendar.)

C#

```
using System;
using System.Globalization;

public class Example
{
    public static void Main()
    {
        var japaneseCal = new JapaneseCalendar();
        var jaJp = new CultureInfo("ja-JP");
        jaJp.DateTimeFormat.Calendar = japaneseCal;

        var date = new DateTime(2, 1, 1, japaneseCal);
        Console.WriteLine($"Gregorian calendar date: {date:d}");
        Console.WriteLine($"Japanese calendar date: {date.ToString("d",
jaJp)}");
    }
}
```

However, if the era changes, the intent of this code becomes ambiguous. Is the date intended to represent the second year of the current era, or is it intended to represent the second year of the Heisei era? There are two ways to avoid this ambiguity:

- Instantiate the date and time value using the default [GregorianCalendar](#) class. You can then use the Japanese calendar or the Japanese Lunisolar calendar for the string representation of dates, as the following example shows.

C#

```
using System;
using System.Globalization;

public class Example
{
    public static void Main()
    {
        var japaneseCal = new JapaneseCalendar();
        var jaJp = new CultureInfo("ja-JP");
        jaJp.DateTimeFormat.Calendar = japaneseCal;

        var date = new DateTime(1905, 2, 12);
    }
}
```

```

Console.WriteLine($"Gregorian calendar date: {date:d}");

        // Call the ToString(IFormatProvider) method.
        Console.WriteLine($"Japanese calendar date: {date.ToString("d",
jaJp)}");

        // Use a FormattableString object.
        FormattableString fmt = $"{date:d}";
        Console.WriteLine($"Japanese calendar date:
{fmt.ToString(jaJp)}");

        // Use the JapaneseCalendar object.
        Console.WriteLine($"Japanese calendar date:
{japCal.DateTimeFormat.GetEraName(japaneseCal.GetEra(date))}" +
$" {japaneseCal.GetYear(date)}/{japaneseCal.GetMonth(date)}/{japaneseCal.G
etDayOfMonth(date)}");

        // Use the current culture.
        CultureInfo.CurrentCulture = jaJp;
        Console.WriteLine($"Japanese calendar date: {date:d}");
    }
}

// The example displays the following output:
//   Gregorian calendar date: 2/12/1905
//   Japanese calendar date: 明治38/2/12

```

- Call a date and time method that explicitly specifies an era. This includes the following methods:
 - The [ToDateTime\(Int32, Int32, Int32, Int32, Int32, Int32, Int32\)](#) method of the [JapaneseCalendar](#) or [JapaneseLunisolarCalendar](#) class.
 - A [DateTime](#) or [DateTimeOffset](#) parsing method, such as [Parse](#), [TryParse](#), [ParseExact](#), or [TryParseExact](#), that includes the string to be parsed and optionally a [DateTimeStyles](#) argument if the current culture is Japanese-Japan ("ja-JP") and that culture's calendar is the [JapaneseCalendar](#). The string to be parsed must include the era.
 - A [DateTime](#) or [DateTimeOffset](#) parsing method that includes a `provider` parameter of type [IFormatProvider](#). `provider` must be either a [CultureInfo](#) object that represents the Japanese-Japan ("ja-JP") culture whose current calendar is [JapaneseCalendar](#) or a [DateTimeFormatInfo](#) object whose [Calendar](#) property is [JapaneseCalendar](#). The string to be parsed must include the era.

The following example uses three of these methods to instantiate a date and time in the Meiji era, which began on September 8, 1868, and ended on July 29, 1912.

```
C#  
  
using System;  
using System.Globalization;  
  
public class Example  
{  
    public static void Main()  
    {  
        var japaneseCal = new JapaneseCalendar();  
        var jaJp = new CultureInfo("ja-JP");  
        jaJp.DateTimeFormat.Calendar = japaneseCal;  
  
        // We can get the era index by calling  
        DateTimeFormatInfo.GetEraName.  
        int eraIndex = 0;  
  
        for (int ctr = 0; ctr <  
jaJp.DateTimeFormat.Calendar.Eras.Length; ctr++)  
            if (jaJp.DateTimeFormat.GetEraName(ctr) == "明治")  
                eraIndex = ctr;  
        var date1 = japaneseCal.ToDateTime(23, 9, 8, 0, 0, 0, 0,  
eraIndex);  
        Console.WriteLine($"{date1.ToString("d", jaJp)} (Gregorian  
{date1:d})");  
  
        try {  
            var date2 = DateTime.Parse("明治23/9/8", jaJp);  
            Console.WriteLine($"{date2.ToString("d", jaJp)} (Gregorian  
{date2:d})");  
        }  
        catch (FormatException)  
        {  
            Console.WriteLine("The parsing operation failed.");  
        }  
  
        try {  
            var date3 = DateTime.ParseExact("明治23/9/8", "gyy/M/d",  
jaJp);  
            Console.WriteLine($"{date3.ToString("d", jaJp)} (Gregorian  
{date3:d})");  
        }  
        catch (FormatException)  
        {  
            Console.WriteLine("The parsing operation failed.");  
        }  
    }  
}  
// The example displays the following output:  
// 明治23/9/8 (Gregorian 9/8/1890)
```

```
// 明治23/9/8 (Gregorian 9/8/1890)
// 明治23/9/8 (Gregorian 9/8/1890)
```

Tip

When working with calendars that support multiple eras, *always* use the Gregorian date to instantiate a date, or specify the era when you instantiate a date and time based on that calendar.

In specifying an era to the [ToDateTime\(Int32, Int32, Int32, Int32, Int32, Int32, Int32, Int32\)](#) method, you provide the index of the era in the calendar's [Eras](#) property. For calendars whose eras are subject to change, however, these indexes are not constant values; the current era is at index 0, and the oldest era is at index `Eras.Length - 1`. When a new era is added to a calendar, the indexes of the previous eras increase by one. You can supply the appropriate era index as follows:

- For dates in the current era, always use the calendar's [CurrentEra](#) property.
- For dates in a specified era, use the [DateTimeFormatInfo.GetEraName](#) method to retrieve the index that corresponds to a specified era name. This requires that the [JapaneseCalendar](#) be the current calendar of the [CultureInfo](#) object that represents the ja-JP culture. (This technique works for the [JapaneseLunisolarCalendar](#) as well, since it supports the same eras as the [JapaneseCalendar](#).) The previous example illustrates this approach.

Calendars, eras, and date ranges: Relaxed range checks

Very much like individual calendars have supported date ranges, eras in the [JapaneseCalendar](#) and [JapaneseLunisolarCalendar](#) classes also have supported ranges. Previously, .NET used strict era range checks to ensure that an era-specific date was within the range of that era. That is, if a date is outside of the range of the specified era, the method throws an [ArgumentOutOfRangeException](#). Currently, .NET uses relaxed ranged checking by default. Updates to all versions of .NET introduced relaxed era range checks; the attempt to instantiate an era-specific date that is outside the range of the specified era "overflows" into the following era, and no exception is thrown.

The following example attempts to instantiate a date in the 65th year of the Showa era, which began on December 25, 1926 and ended on January 7, 1989. This date corresponds to January 9, 1990, which is outside the range of the Showa era in the [JapaneseCalendar](#). As the output from the example illustrates, the date displayed by the example is January 9, 1990, in the second year of the Heisei era.

C#

```
using System;
using System.Globalization;

public class Example
{
    public static void Main()
    {
        var jaJp = new CultureInfo("ja-JP");
        var cal = new JapaneseCalendar();
        jaJp.DateTimeFormat.Calendar = cal;
        string showaEra = "昭和";

        var dt = cal.ToDateTime(65, 1, 9, 15, 0, 0, 0, GetEraIndex(showaEra));
        FormattableString fmt = $"{dt:d}";

        Console.WriteLine($"Japanese calendar date: {fmt.ToString(jaJp)}");
        Console.WriteLine($"Gregorian calendar date: {fmt}");

        int GetEraIndex(string eraName)
        {
            foreach (var ctr in cal.Eras)
                if (jaJp.DateTimeFormat.GetEraName(ctr) == eraName)
                    return ctr;

            return 0;
        }
    }
}
// The example displays the following output:
//   Japanese calendar date: 平成2/1/9
//   Gregorian calendar date: 1/9/1990
```

If relaxed range checks are undesirable, you can restore strict range checks in a number of ways, depending on the version of .NET on which your application is running:

- **.NET Core:** Add the following to the `.netcore.runtime.json` config file:

JSON

```
"runtimeOptions": {
    "configProperties": {
        "Switch.System.Globalization.EnforceJapaneseEraYearRanges": true
    }
}
```

- **.NET Framework 4.6 or later:** Set the following `AppContext` switch in the `app.config` file:

XML

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <runtime>
    <AppContextSwitchOverrides
      value="Switch.System.Globalization.EnforceJapaneseEraYearRanges=true"
    />
  </runtime>
</configuration>
```

- **.NET Framework 4.5.2 or earlier:** Set the following registry value:

| Value | |
|--------------|--|
| Key | HKEY_LOCAL_MACHINE\Software\Microsoft\.NETFramework\AppContext |
| Entry | Switch.System.Globalization.EnforceJapaneseEraYearRanges |
| Type | REG_SZ |
| Value | true |

With strict range checks enabled, the previous example throws an [ArgumentException](#) and displays the following output:

Console

```
Unhandled Exception: System.ArgumentOutOfRangeException: Valid values are
between 1 and 64, inclusive.
Parameter name: year
  at System.Globalization.GregorianCalendarHelper.GetYearOffset(Int32 year,
Int32 era, Boolean throwOnError)
  at System.Globalization.GregorianCalendarHelper.ToDateTime(Int32 year,
Int32 month, Int32 day, Int32 hour, Int32 minute, Int32 second, Int32
millisecond, Int32 era)
  at Example.Main()
```

Represent dates in calendars with multiple eras

If a [Calendar](#) object supports eras and is the current calendar of a [CultureInfo](#) object, the era is included in the string representation of a date and time value for the full date and time, long date, and short date patterns. The following example displays these date patterns when the current culture is Japan (Japanese) and the current calendar is the Japanese calendar.

C#

```

using System;
using System.Globalization;
using System.IO;
using System.Threading;

public class Example
{
    public static void Main()
    {
        StreamWriter sw = new StreamWriter(@".\eras.txt");
        DateTime dt = new DateTime(2012, 5, 1);

        CultureInfo culture = CultureInfo.CreateSpecificCulture("ja-JP");
        DateTimeFormatInfo dtfi = culture.DateTimeFormat;
        dtfi.Calendar = new JapaneseCalendar();
        Thread.CurrentThread.CurrentCulture = culture;

        sw.WriteLine("\n{0,-43} {1}", "Full Date and Time Pattern:",
        dtfi.FullDateTimePattern);
        sw.WriteLine(dt.ToString("F"));
        sw.WriteLine();

        sw.WriteLine("\n{0,-43} {1}", "Long Date Pattern:",
        dtfi.LongDatePattern);
        sw.WriteLine(dt.ToString("D"));

        sw.WriteLine("\n{0,-43} {1}", "Short Date Pattern:",
        dtfi.ShortDatePattern);
        sw.WriteLine(dt.ToString("d"));
        sw.Close();
    }
}

// The example writes the following output to a file:
//   Full Date and Time Pattern:          gg y'年'M'月'd'日' H:mm:ss
//   平成 24年5月1日 0:00:00
//
//   Long Date Pattern:                  gg y'年'M'月'd'日'
//   平成 24年5月1日
//
//   Short Date Pattern:                gg y/M/d
//   平成 24/5/1

```

⚠ Warning

The **JapaneseCalendar** class is the only calendar class in .NET that both supports dates in more than one era and that can be the current calendar of a **CultureInfo** object - specifically, of a **CultureInfo** object that represents the Japanese (Japan) culture.

For all calendars, the "g" custom format specifier includes the era in the result string. The following example uses the "MM-dd-yyyy g" custom format string to include the era in the result string when the current calendar is the Gregorian calendar.

C#

```
DateTime dat = new DateTime(2012, 5, 1);
Console.WriteLine("{0:MM-dd-yyyy g}", dat);
// The example displays the following output:
//      05-01-2012 A.D.
```

In cases where the string representation of a date is expressed in a calendar that is not the current calendar, the [Calendar](#) class includes a [Calendar.GetEra](#) method that can be used along with the [Calendar.GetYear](#), [Calendar.GetMonth](#), and [Calendar.GetDayOfMonth](#) methods to unambiguously indicate a date as well as the era to which it belongs. The following example uses the [JapaneseLunisolarCalendar](#) class to provide an illustration. However, note that including a meaningful name or abbreviation instead of an integer for the era in the result string requires that you instantiate a [DateTimeFormatInfo](#) object and make [JapaneseCalendar](#) its current calendar. (The [JapaneseLunisolarCalendar](#) calendar cannot be the current calendar of any culture, but in this case the two calendars share the same eras.)

C#

```
using System;
using System.Globalization;

public class Example
{
    public static void Main()
    {
        DateTime date1 = new DateTime(2011, 8, 28);
        Calendar cal = new JapaneseLunisolarCalendar();

        Console.WriteLine("{0} {1:d4}/{2:d2}/{3:d2}",
                         cal.GetEra(date1),
                         cal.GetYear(date1),
                         cal.GetMonth(date1),
                         cal.GetDayOfMonth(date1));

        // Display eras
        CultureInfo culture = CultureInfo.CreateSpecificCulture("ja-JP");
        DateTimeFormatInfo dtfi = culture.DateTimeFormat;
        dtfi.Calendar = new JapaneseCalendar();

        Console.WriteLine("{0} {1:d4}/{2:d2}/{3:d2}",
                         dtfi.GetAbbreviatedEraName(cal.GetEra(date1)),
                         cal.GetYear(date1),
                         cal.GetMonth(date1),
```

```

        cal.GetDayOfMonth(date1));
    }
}

// The example displays the following output:
//      4 0023/07/29
//      平 0023/07/29

```

In the Japanese calendars, the first year of an era is called Gannen (元年). For example, instead of Heisei 1, the first year of the Heisei era can be described as Heisei Gannen. .NET adopts this convention in formatting operations for dates and times formatted with the following standard or custom date and time format strings when they are used with a [CultureInfo](#) object that represents the Japanese-Japan ("ja-JP") culture with the [JapaneseCalendar](#) class:

- [The long date pattern](#), indicated by the "D" standard date and time format string.
- [The full date long time pattern](#), indicated by the "F" standard date and time format string.
- [The full date short time pattern](#), indicated by the "f" standard date and time format string.
- [The year/month pattern](#), indicated by the "Y" or "y" standard date and time format string.
- The "ggy'年'" or "ggy年" [custom date and time format string](#).

For example, the following example displays a date in the first year of the Heisei era in the [JapaneseCalendar](#).

C#

```

using System;
using System.Globalization;

public class Example
{
    public static void Main()
    {
        var enUs = new CultureInfo("en-US");
        var japaneseCal = new JapaneseCalendar();
        var jaJp = new CultureInfo("ja-JP");
        jaJp.DateTimeFormat.Calendar = japaneseCal;
        string heiseiEra = "平成";

        var date = japaneseCal.ToDateTime(1, 8, 18, 0, 0, 0, 0,
GetEraIndex(heiseiEra));
        FormattableString fmt = $"{date:D}";
        Console.WriteLine($"Japanese calendar date: {fmt.ToString(jaJp)}
(Gregorian: {fmt.ToString(enUs)})");

        int GetEraIndex(string eraName)
    }
}

```

```

    {
        foreach (var ctr in japaneseCal.Eras)
            if (jaJp.DateTimeFormat.GetEraName(ctr) == eraName)
                return ctr;

        return 0;
    }
}

// The example displays the following output:
//   Japanese calendar date: 平成元年8月18日 (Gregorian: Friday, August 18,
1989)

```

If this behavior is undesirable in formatting operations, you can restore the previous behavior, which always represents the first year of an era as "1" rather than "Gannen", by doing the following, depending on the version of .NET:

- **.NET Core:** Add the following to the `.netcore.runtime.json` config file:

JSON

```

"runtimeOptions": {
    "configProperties": {
        "Switch.System.Globalization.FormatJapaneseFirstYearAsANumber": true
    }
}

```

- **.NET Framework 4.6 or later:** Set the following `AppContext` switch in the `app.config` file:

XML

```

<?xml version="1.0" encoding="utf-8"?>
<configuration>
    <runtime>
        <AppContextSwitchOverrides
value="Switch.System.Globalization.FormatJapaneseFirstYearAsANumber=true" />
    </runtime>
</configuration>

```

- **.NET Framework 4.5.2 or earlier:** Set the following registry value:

| | Value |
|------------|--|
| Key | HKEY_LOCAL_MACHINE\Software\Microsoft\.NETFramework\AppContext |

| Value | |
|--------------|--|
| Entry | Switch.System.Globalization.FormatJapaneseFirstYearAsANumber |
| Type | REG_SZ |
| Value | true |

With gannen support in formatting operations disabled, the previous example displays the following output:

Console

```
Japanese calendar date: 平成1年8月18日 (Gregorian: Friday, August 18, 1989)
```

.NET has also been updated so that date and time parsing operations support strings that contain the year represented as either "1" or Gannen. Although you should not need to do this, you can restore the previous behavior to recognize only "1" as the first year of an era. You can do this as follows, depending on the version of .NET:

- **.NET Core:** Add the following to the `.netcore.runtime.json` config file:

JSON

```
"runtimeOptions": {
  "configProperties": {
    "Switch.System.Globalization.EnforceLegacyJapaneseDateParsing": true
  }
}
```

- **.NET Framework 4.6 or later:** Set the following `AppContext` switch in the `app.config` file:

XML

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <runtime>
    <AppContextSwitchOverrides
      value="Switch.System.Globalization.EnforceLegacyJapaneseDateParsing=true"
    />
  </runtime>
</configuration>
```

- **.NET Framework 4.5.2 or earlier:** Set the following registry value:

| Value | |
|-------|---|
| Key | HKEY_LOCAL_MACHINE\Software\Microsoft\.NETFramework\ ApplicationContext |
| Entry | Switch.System.Globalization.EnforceLegacyJapaneseDateParsing |
| Type | REG_SZ |
| Value | true |

See also

- [How to: Display dates in non-Gregorian calendars](#)
- [Calendar class](#)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

How to use the DateOnly and TimeOnly structures

Article • 01/12/2023

The [DateOnly](#) and [TimeOnly](#) structures were introduced with .NET 6 and represent a specific date or time-of-day, respectively. Prior to .NET 6, and always in .NET Framework, developers used the [DateTime](#) type (or some other alternative) to represent one of the following:

- A whole date and time.
- A date, disregarding the time.
- A time, disregarding the date.

`DateOnly` and `TimeOnly` are types that represent those particular portions of a `DateTime` type.

ⓘ Important

[DateOnly](#) and [TimeOnly](#) types aren't available in .NET Framework.

The DateOnly structure

The [DateOnly](#) structure represents a specific date, without time. Since it has no time component, it represents a date from the start of the day to the end of the day. This structure is ideal for storing specific dates, such as a birth date, an anniversary date, or business-related dates.

Although you could use `DateTime` while ignoring the time component, there are a few benefits to using `DateOnly` over `DateTime`:

- The `DateTime` structure may roll into the previous or next day if it's offset by a time zone. `DateOnly` can't be offset by a time zone, and it always represents the date that was set.
- Serializing a `DateTime` structure includes the time component, which may obscure the intent of the data. Also, `DateOnly` serializes less data.
- When code interacts with a database, such as SQL Server, whole dates are generally stored as the `date` data type, which doesn't include a time. `DateOnly`

matches the database type better.

`DateOnly` has a range from 0001-01-01 through 9999-12-31, just like `DateTime`. You can specify a specific calendar in the `DateOnly` constructor. However, a `DateOnly` object always represents a date in the proleptic Gregorian calendar, regardless of which calendar was used to construct it. For example, you can build the date from a Hebrew calendar, but the date is converted to Gregorian:

C#

```
var hebrewCalendar = new System.Globalization.HebrewCalendar();
var theDate = new DateOnly(5776, 2, 8, hebrewCalendar); // 8 Cheshvan 5776

Console.WriteLine(theDate);

/* This example produces the following output:
 *
 * 10/21/2015
 */
```

DateOnly examples

Use the following examples to learn about `DateOnly`:

- [Convert DateTime to DateOnly](#)
- [Add or subtract days, months, years](#)
- [Parse and format DateOnly](#)
- [Compare DateOnly](#)

Convert DateTime to DateOnly

Use the `DateOnly.FromDateTime` static method to create a `DateOnly` type from a `DateTime` type, as demonstrated in the following code:

C#

```
var today = DateOnly.FromDateTime(DateTime.Now);
Console.WriteLine($"Today is {today}");

/* This example produces output similar to the following:
 *
 * Today is 12/28/2022
 */
```

Add or subtract days, months, years

There are three methods used to adjust a [DateOnly](#) structure: [AddDays](#), [AddMonths](#), and [AddYears](#). Each method takes an integer parameter, and increases the date by that measurement. If a negative number is provided, the date is decreased by that measurement. The methods return a new instance of [DateOnly](#), as the structure is immutable.

C#

```
var theDate = new DateOnly(2015, 10, 21);

var nextDay = theDate.AddDays(1);
var previousDay = theDate.AddDays(-1);
var decadeLater = theDate.AddYears(10);
var lastMonth = theDate.AddMonths(-1);

Console.WriteLine($"Date: {theDate}");
Console.WriteLine($" Next day: {nextDay}");
Console.WriteLine($" Previous day: {previousDay}");
Console.WriteLine($" Decade later: {decadeLater}");
Console.WriteLine($" Last month: {lastMonth}");

/* This example produces the following output:
 *
 * Date: 10/21/2015
 * Next day: 10/22/2015
 * Previous day: 10/20/2015
 * Decade later: 10/21/2025
 * Last month: 9/21/2015
 */
```

Parse and format DateOnly

[DateOnly](#) can be parsed from a string, just like the [DateTime](#) structure. All of the standard .NET date-based parsing tokens work with [DateOnly](#). When converting a [DateOnly](#) type to a string, you can use standard .NET date-based formatting patterns too. For more information about formatting strings, see [Standard date and time format strings](#).

C#

```
var theDate = DateOnly.ParseExact("21 Oct 2015", "dd MMM yyyy",
CultureInfo.InvariantCulture); // Custom format
var theDate2 = DateOnly.Parse("October 21, 2015",
CultureInfo.InvariantCulture);
```

```

Console.WriteLine(theDate.ToString("m", CultureInfo.InvariantCulture));
// Month day pattern
Console.WriteLine(theDate2.ToString("o", CultureInfo.InvariantCulture));
// ISO 8601 format
Console.WriteLine(theDate2.ToString("o"));

/* This example produces the following output:
 *
 * October 21
 * 2015-10-21
 * Wednesday, October 21, 2015
 */

```

Compare DateOnly

[DateOnly](#) can be compared with other instances. For example, you can check if a date is before or after another, or if a date today matches a specific date.

C#

```

var theDate = DateOnly.ParseExact("21 Oct 2015", "dd MMM yyyy",
CultureInfo.InvariantCulture); // Custom format
var theDate2 = DateOnly.Parse("October 21, 2015",
CultureInfo.InvariantCulture);
var dateLater = theDate.AddMonths(6);
var dateBefore = theDate.AddDays(-10);

Console.WriteLine($"Consider {theDate}...");
Console.WriteLine($" Is '{nameof(theDate2)}' equal? {theDate == theDate2}");
Console.WriteLine($" Is {dateLater} after? {dateLater > theDate} ");
Console.WriteLine($" Is {dateLater} before? {dateLater < theDate} ");
Console.WriteLine($" Is {dateBefore} after? {dateBefore > theDate} ");
Console.WriteLine($" Is {dateBefore} before? {dateBefore < theDate} ");

/* This example produces the following output:
 *
 * Consider 10/21/2015
 * Is 'theDate2' equal? True
 * Is 4/21/2016 after? True
 * Is 4/21/2016 before? False
 * Is 10/11/2015 after? False
 * Is 10/11/2015 before? True
 */

```

The TimeOnly structure

The [TimeOnly](#) structure represents a time-of-day value, such as a daily alarm clock or what time you eat lunch each day. `TimeOnly` is limited to the range of `00:00:00.0000000`

- 23:59:59.9999999, a specific time of day.

Prior to the `TimeOnly` type being introduced, programmers typically used either the `DateTime` type or the `TimeSpan` type to represent a specific time. However, using these structures to simulate a time without a date may introduce some problems, which `TimeOnly` solves:

- `TimeSpan` represents elapsed time, such as time measured with a stopwatch. The upper range is more than 29,000 years, and its value can be negative to indicate moving backwards in time. A negative `TimeSpan` doesn't indicate a specific time of the day.
- If `TimeSpan` is used as a time of day, there's a risk that it could be manipulated to a value outside of the 24-hour day. `TimeOnly` doesn't have this risk. For example, if an employee's work shift starts at 18:00 and lasts for 8 hours, adding 8 hours to the `TimeOnly` structure rolls over to 2:00
- Using `DateTime` for a time of day requires that an arbitrary date be associated with the time, and then later disregarded. It's common practice to choose `DateTime.MinValue` (0001-01-01) as the date, however, if hours are subtracted from the `DateTime` value, an `OutOfRange` exception might occur. `TimeOnly` doesn't have this problem as the time rolls forwards and backwards around the 24-hour timeframe.
- Serializing a `DateTime` structure includes the date component, which may obscure the intent of the data. Also, `TimeOnly` serializes less data.

TimeOnly examples

Use the following examples to learn about `TimeOnly`:

- [Convert DateTime to TimeOnly](#)
- [Add or subtract time](#)
- [Parse and format TimeOnly](#)
- [Work with TimeSpan and DateTime](#)
- [Arithmetic operators and comparing TimeOnly](#)

Convert DateTime to TimeOnly

Use the `TimeOnly.FromDateTime` static method to create a `TimeOnly` type from a `DateTime` type, as demonstrated in the following code:

C#

```
var now = TimeOnly.FromDateTime(DateTime.Now);
Console.WriteLine($"It is {now} right now");

/* This example produces output similar to the following:
 *
 * It is 2:01 PM right now
 */
```

Add or subtract time

There are three methods used to adjust a `TimeOnly` structure: `AddHours`, `AddMinutes`, and `Add`. Both `AddHours` and `AddMinutes` take an integer parameter, and adjust the value accordingly. You can use a negative value to subtract and a positive value to add. The methods return a new instance of `TimeOnly` is returned, as the structure is immutable. The `Add` method takes a `TimeSpan` parameter and adds or subtracts the value from the `TimeOnly` value.

Because `TimeOnly` only represents a 24-hour period, it rolls over forwards or backwards appropriately when adding values supplied to those three methods. For example, if you use a value of `01:30:00` to represent 1:30 AM, then add -4 hours from that period, it rolls backwards to `21:30:00`, which is 9:30 PM. There are method overloads for `AddHours`, `AddMinutes`, and `Add` that capture the number of days rolled over.

C#

```
var theTime = new TimeOnly(7, 23, 11);

var hourLater = theTime.AddHours(1);
var minutesBefore = theTime.AddMinutes(-12);
var secondsAfter = theTime.Add(TimeSpan.FromSeconds(10));
var daysLater = theTime.Add(new TimeSpan(hours: 21, minutes: 200, seconds: 83), out int wrappedDays);
var daysBehind = theTime.AddHours(-222, out int wrappedDaysFromHours);

Console.WriteLine($"Time: {theTime}");
Console.WriteLine($" Hours later: {hourLater}");
Console.WriteLine($" Minutes before: {minutesBefore}");
Console.WriteLine($" Seconds after: {secondsAfter}");
Console.WriteLine($" {daysLater} is the time, which is {wrappedDays} days later");
Console.WriteLine($" {daysBehind} is the time, which is {wrappedDaysFromHours} days prior");

/* This example produces the following output:
 *
```

```
* Time: 7:23 AM
* Hours later: 8:23 AM
* Minutes before: 7:11 AM
* Seconds after: 7:23 AM
* 7:44 AM is the time, which is 1 days later
* 1:23 AM is the time, which is -9 days prior
*/
```

Parse and format `TimeOnly`

`TimeOnly` can be parsed from a string, just like the `DateTime` structure. All of the standard .NET time-based parsing tokens work with `TimeOnly`. When converting a `TimeOnly` type to a string, you can use standard .NET date-based formatting patterns too. For more information about formatting strings, see [Standard date and time format strings](#).

C#

```
var theTime = TimeOnly.ParseExact("5:00 pm", "h:mm tt",
CultureInfo.InvariantCulture); // Custom format
var theTime2 = TimeOnly.Parse("17:30:25", CultureInfo.InvariantCulture);

Console.WriteLine(theTime.ToString("o", CultureInfo.InvariantCulture));
// Round-trip pattern.
Console.WriteLine(theTime2.ToString("t", CultureInfo.InvariantCulture));
// Long time format
Console.WriteLine(theTime2.ToLongTimeString());

/* This example produces the following output:
*
* 17:00:00.0000000
* 17:30
* 5:30:25 PM
*/
```

Serialize `DateOnly` and `TimeOnly` types

With .NET 7+, `System.Text.Json` supports serializing and deserializing `DateOnly` and `TimeOnly` types. Consider the following object:

C#

```
sealed file record Appointment(
    Guid Id,
    string Description,
    DateOnly Date,
```

```
TimeOnly StartTime,  
TimeOnly EndTime);
```

The following example serializes an `Appointment` object, displays the resulting JSON, and then deserializes it back into a new instance of the `Appointment` type. Finally, the original and newly deserialized instances are compared for equality and the results are written to the console:

C#

```
Appointment originalAppointment = new(  
    Id: Guid.NewGuid(),  
    Description: "Take dog to veterinarian.",  
    Date: new DateOnly(2002, 1, 13),  
    StartTime: new TimeOnly(5,15),  
    EndTime: new TimeOnly(5, 45));  
string serialized = JsonSerializer.Serialize(originalAppointment);  
  
Console.WriteLine($"Resulting JSON: {serialized}");  
  
Appointment deserializedAppointment =  
    JsonSerializer.Deserialize<Appointment>(serialized)!;  
  
bool valuesAreTheSame = originalAppointment == deserializedAppointment;  
Console.WriteLine($"""  
    Original record has the same values as the deserialized record:  
{valuesAreTheSame}  
"");
```

In the preceding code:

- An `Appointment` object is instantiated and assigned to the `appointment` variable.
- The `appointment` instance is serialized to JSON using `JsonSerializer.Serialize`.
- The resulting JSON is written to the console.
- The JSON is deserialized back into a new instance of the `Appointment` type using `JsonSerializer.Deserialize`.
- The original and newly deserialized instances are compared for equality.
- The result of the comparison is written to the console.

For more information, see [How to serialize and deserialize JSON in .NET](#).

Work with TimeSpan and DateTime

`TimeOnly` can be created from and converted to a `TimeSpan`. Also, `TimeOnly` can be used with a `DateTime`, either to create the `TimeOnly` instance, or to create a `DateTime` instance as long as a date is provided.

The following example creates a `TimeOnly` object from a `TimeSpan`, and then converts it back:

C#

```
// TimeSpan must be in the range of 00:00:00.0000000 to 23:59:59.9999999
var theTime = TimeOnly.FromTimeSpan(new TimeSpan(23, 59, 59));
var theTimeSpan = theTime.ToDateTime();

Console.WriteLine($"Variable '{nameof(theTime)}' is {theTime}");
Console.WriteLine($"Variable '{nameof(theTimeSpan)}' is {theTimeSpan}");

/* This example produces the following output:
 *
 * Variable 'theTime' is 11:59 PM
 * Variable 'theTimeSpan' is 23:59:59
 */
```

The following example creates a `DateTime` from a `TimeOnly` object, with an arbitrary date chosen:

C#

```
var theTime = new TimeOnly(11, 25, 46); // 11:25 PM and 46 seconds
var theDate = new DateOnly(2015, 10, 21); // October 21, 2015
var theDateTime = theDate.ToDateTime(theTime);
var reverseTime = TimeOnly.FromDateTime(theDateTime);

Console.WriteLine($"Date only is {theDate}");
Console.WriteLine($"Time only is {theTime}");
Console.WriteLine();
Console.WriteLine($"Combined to a DateTime type, the value is
{theDateTime}");
Console.WriteLine($"Converted back from DateTime, the time is
{reverseTime}");

/* This example produces the following output:
 *
 * Date only is 10/21/2015
 * Time only is 11:25 AM
 *
 * Combined to a DateTime type, the value is 10/21/2015 11:25:46 AM
 * Converted back from DateTime, the time is 11:25 AM
 */
```

Arithmetic operators and comparing `TimeOnly`

Two `TimeOnly` instances can be compared with one another, and you can use the `IsBetween` method to check if a time is between two other times. When an addition or

subtraction operator is used on a `TimeOnly`, a `TimeSpan` is returned, representing a duration of time.

C#

```
var start = new TimeOnly(10, 12, 01); // 10:12:01 AM
var end = new TimeOnly(14, 00, 53); // 02:00:53 PM

var outside = start.AddMinutes(-3);
var inside = start.AddMinutes(120);

Console.WriteLine($"Time starts at {start} and ends at {end}");
Console.WriteLine($" Is {outside} between the start and end?
{outside.IsBetween(start, end)}");
Console.WriteLine($" Is {inside} between the start and end?
{inside.IsBetween(start, end)}");
Console.WriteLine($" Is {start} less than {end}? {start < end}");
Console.WriteLine($" Is {start} greater than {end}? {start > end}");
Console.WriteLine($" Does {start} equal {end}? {start == end}");
Console.WriteLine($" The time between {start} and {end} is {end - start}");

/* This example produces the following output:
 *
 * Time starts at 10:12 AM and ends at 2:00 PM
 * Is 10:09 AM between the start and end? False
 * Is 12:12 PM between the start and end? True
 * Is 10:12 AM less than 2:00 PM? True
 * Is 10:12 AM greater than 2:00 PM? False
 * Does 10:12 AM equal 2:00 PM? False
 * The time between 10:12 AM and 2:00 PM is 03:48:52
*/
```

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback [here](#).

 [Open a documentation issue](#)

 [Provide product feedback](#)

Performing arithmetic operations with dates and times

Article • 09/15/2021

Although both the [DateTime](#) and the [DateTimeOffset](#) structures provide members that perform arithmetic operations on their values, the results of arithmetic operations are very different. This article examines those differences, relates them to degrees of time zone awareness in date and time data, and discusses how to perform fully time zone aware operations using date and time data.

Comparisons and arithmetic operations with DateTime values

The [DateTime.Kind](#) property allows a [DateTimeKind](#) value to be assigned to the date and time to indicate whether it represents local time, Coordinated Universal Time (UTC), or the time in an unspecified time zone. However, this limited time zone information is ignored when comparing or performing date and time arithmetic on [DateTimeKind](#) values. The following example, which compares the current local time with the current UTC time, illustrates how the time zone information is ignored.

C#

```
using System;

public enum TimeComparison
{
    EarlierThan = -1,
    TheSameAs = 0,
    LaterThan = 1
}

public class DateManipulation
{
    public static void Main()
    {
        DateTime localTime = DateTime.Now;
        DateTime utcTime = DateTime.UtcNow;

        Console.WriteLine("Difference between {0} and {1} time: {2}:{3}
hours",
            localTime.Kind,
            utcTime.Kind,
            (localTime - utcTime).Hours,
            (localTime - utcTime).Minutes);
    }
}
```

```

        Console.WriteLine("The {0} time is {1} the {2} time.",
                           localTime.Kind,
                           Enum.GetName(typeof(TimeComparison),
                           localTime.CompareTo(utcTime)),
                           utcTime.Kind);
    }
}

// If run in the U.S. Pacific Standard Time zone, the example displays
// the following output to the console:
//   Difference between Local and Utc time: -7:0 hours
//   The Local time is EarlierThan the Utc time.

```

The [CompareTo\(DateTime\)](#) method reports that the local time is earlier than (or less than) the UTC time, and the subtraction operation indicates that the difference between UTC and the local time for a system in the U.S. Pacific Standard Time zone is seven hours. But because these two values provide different representations of a single point in time, it's clear in this case that the time interval is completely attributable to the local time zone's offset from UTC.

More generally, the [DateTime.Kind](#) property does not affect the results returned by [Kind](#) comparison and arithmetic methods (as the comparison of two identical points in time indicates), although it can affect the interpretation of those results. For example:

- The result of any arithmetic operation performed on two date and time values whose [DateTime.Kind](#) properties both equal [DateTimeKind](#) reflects the actual time interval between the two values. Similarly, the comparison of two such date and time values accurately reflects the relationship between times.
- The result of any arithmetic or comparison operation performed on two date and time values whose [DateTime.Kind](#) properties both equal [DateTimeKind](#) or on two date and time values with different [DateTime.Kind](#) property values reflects the difference in clock time between the two values.
- Arithmetic or comparison operations on local date and time values do not consider whether a particular value is ambiguous or invalid, nor do they take account of the effect of any adjustment rules that result from the local time zone's transition to or from daylight saving time.
- Any operation that compares or calculates the difference between UTC and a local time includes a time interval equal to the local time zone's offset from UTC in the result.
- Any operation that compares or calculates the difference between an unspecified time and either UTC or the local time reflects simple clock time. Time zone

differences are not considered, and the result does not reflect the application of time zone adjustment rules.

- Any operation that compares or calculates the difference between two unspecified times may include an unknown interval that reflects the difference between the time in two different time zones.

There are many scenarios in which time zone differences do not affect date and time calculations (for a discussion of some of these scenarios, see [Choosing between DateTime, DateTimeOffset, TimeSpan, and TimeZoneInfo](#)) or in which the context of the date and time data defines the meaning of comparison or arithmetic operations.

Comparisons and arithmetic operations with DateTimeOffset values

A [DateTimeOffset](#) value includes not only a date and time, but also an offset that unambiguously defines that date and time relative to UTC. This offset makes it possible to define equality differently than for [DateTime](#) values. Whereas [DateTime](#) values are equal if they have the same date and time value, [DateTimeOffset](#) values are equal if they both refer to the same point in time. When used in comparisons and in most arithmetic operations that determine the interval between two dates and times, a [DateTimeOffset](#) value is more accurate and less in need of interpretation. The following example, which is the [DateTimeOffset](#) equivalent to the previous example that compared local and UTC [DateTimeOffset](#) values, illustrates this difference in behavior.

```
C#  
  
using System;  
  
public enum TimeComparison  
{  
    EarlierThan = -1,  
    TheSameAs = 0,  
    LaterThan = 1  
}  
  
public class DateTimeOffsetManipulation  
{  
    public static void Main()  
    {  
        DateTimeOffset localTime = DateTimeOffset.Now;  
        DateTimeOffset utcTime = DateTimeOffset.UtcNow;  
  
        Console.WriteLine("Difference between local time and UTC: {0}:{1:D2}  
hours",  
            (localTime - utcTime).Hours,
```

```

        (localTime - utcTime).Minutes);
    Console.WriteLine("The local time is {0} UTC.",
        Enum.GetName(typeof(TimeComparison),
    localTime.CompareTo(utcTime)));
}
}
// Regardless of the local time zone, the example displays
// the following output to the console:
//     Difference between local time and UTC: 0:00 hours.
//     The local time is TheSameAs UTC.

```

In this example, the `CompareTo` method indicates that the current local time and the current UTC time are equal, and subtraction of `CompareTo(DateTimeOffset)` values indicates that the difference between the two times is `TimeSpan.Zero`.

The chief limitation of using `DateTimeOffset` values in date and time arithmetic is that although `DateTimeOffset` values have some time zone awareness, they are not fully time zone aware. Although the `DateTimeOffset` value's offset reflects a time zone's offset from UTC when a `DateTimeOffset` variable is first assigned a value, it becomes disassociated from the time zone thereafter. Because it is no longer directly associated with an identifiable time, the addition and subtraction of date and time intervals does not consider a time zone's adjustment rules.

To illustrate, the transition to daylight saving time in the U.S. Central Standard Time zone occurs at 2:00 A.M. on March 9, 2008. With that in mind, adding a two and a half hour interval to a Central Standard time of 1:30 A.M. on March 9, 2008, should produce a date and time of 5:00 A.M. on March 9, 2008. However, as the following example shows, the result of the addition is 4:00 A.M. on March 9, 2008. The result of this operation does represent the correct point in time, although it is not the time in the time zone in which we are interested (that is, it does not have the expected time zone offset).

C#

```

using System;

public class IntervalArithmetic
{
    public static void Main()
    {
        DateTime generalTime = new DateTime(2008, 3, 9, 1, 30, 0);
        const string tzName = "Central Standard Time";
        TimeSpan twoAndAHalfHours = new TimeSpan(2, 30, 0);

        // Instantiate DateTimeOffset value to have correct CST offset
        try
        {
            DateTimeOffset centralTime1 = new DateTimeOffset(generalTime,

```

```

TimeZoneInfo.FindSystemTimeZoneById(tzName).GetUtcOffset(generalTime));

    // Add two and a half hours
    DateTimeOffset centralTime2 = centralTime1.Add(twoAndAHalfHours);
    // Display result
    Console.WriteLine("{0} + {1} hours = {2}", centralTime1,
                      twoAndAHalfHours.ToString(),
                      centralTime2);
}
catch (TimeZoneNotFoundException)
{
    Console.WriteLine("Unable to retrieve Central Standard Time zone
information.");
}
}

// The example displays the following output to the console:
//      3/9/2008 1:30:00 AM -06:00 + 02:30:00 hours = 3/9/2008 4:00:00 AM
-06:00

```

Arithmetic operations with times in time zones

The [TimeZoneInfo](#) class includes conversion methods that automatically apply adjustments when they convert times from one time zone to another. These conversion methods include:

- The [ConvertTime](#) and [ConvertTimeBySystemTimeZoneId](#) methods, which convert times between any two time zones.
- The [ConvertTimeFromUtc](#) and [ConvertTimeToUtc](#) methods, which convert the time in a particular time zone to UTC, or convert UTC to the time in a particular time zone.

For details, see [Converting times between time zones](#).

The [TimeZoneInfo](#) class does not provide any methods that automatically apply adjustment rules when you perform date and time arithmetic. However, you can apply adjustment rules by converting the time in a time zone to UTC, performing the arithmetic operation, and then converting from UTC back to the time in the time zone.

For details, see [How to: Use time zones in date and time arithmetic](#).

For example, the following code is similar to the previous code that added two-and-a-half hours to 2:00 A.M. on March 9, 2008. However, because it converts a Central Standard time to UTC before it performs date and time arithmetic, and then converts the

result from UTC back to Central Standard time, the resulting time reflects the Central Standard Time Zone's transition to daylight saving time.

```
C#  
  
using System;  
  
public class TimeZoneAwareArithmetic  
{  
    public static void Main()  
    {  
        const string tzName = "Central Standard Time";  
  
        DateTime generalTime = new DateTime(2008, 3, 9, 1, 30, 0);  
        TimeZoneInfo cst = TimeZoneInfo.FindSystemTimeZoneById(tzName);  
        TimeSpan twoAndAHalfHours = new TimeSpan(2, 30, 0);  
  
        // Instantiate DateTimeOffset value to have correct CST offset  
        try  
        {  
            DateTimeOffset centralTime1 = new DateTimeOffset(generalTime,  
                cst.GetUtcOffset(generalTime));  
  
            // Add two and a half hours  
            DateTimeOffset utcTime = centralTime1.ToUniversalTime();  
            utcTime += twoAndAHalfHours;  
  
            DateTimeOffset centralTime2 = TimeZoneInfo.ConvertTime(utcTime,  
cst);  
            // Display result  
            Console.WriteLine("{0} + {1} hours = {2}", centralTime1,  
twoAndAHalfHours.ToString(),  
                           centralTime2);  
        }  
        catch (TimeZoneNotFoundException)  
        {  
            Console.WriteLine("Unable to retrieve Central Standard Time zone  
information.");  
        }  
    }  
}  
// The example displays the following output to the console:  
//      3/9/2008 1:30:00 AM -06:00 + 02:30:00 hours = 3/9/2008 5:00:00 AM  
-05:00
```

See also

- [Dates, times, and time zones](#)
- [How to: Use time zones in date and time arithmetic](#)

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

DateTime and DateTimeOffset support in System.Text.Json

Article • 01/12/2023

The `System.Text.Json` library parses and writes `DateTime` and `DateTimeOffset` values according to the ISO 8601-1:2019 extended profile. `Converters` provide custom support for serializing and deserializing with `JsonSerializer`. You can also use `Utf8JsonReader` and `Utf8JsonWriter` to implement custom support.

Support for the ISO 8601-1:2019 format

The `JsonSerializer`, `Utf8JsonReader`, `Utf8JsonWriter`, and `JsonElement` types parse and write `DateTime` and `DateTimeOffset` text representations according to the extended profile of the ISO 8601-1:2019 format. For example, `2019-07-26T16:59:57-05:00`.

`DateTime` and `DateTimeOffset` data can be serialized with `JsonSerializer`:

C#

```
using System.Text.Json;

public class Example
{
    private class Product
    {
        public string? Name { get; set; }
        public DateTime ExpiryDate { get; set; }
    }

    public static void Main(string[] args)
    {
        Product p = new Product();
        p.Name = "Banana";
        p.ExpiryDate = new DateTime(2019, 7, 26);

        string json = JsonSerializer.Serialize(p);
        Console.WriteLine(json);
    }
}

// The example displays the following output:
// {"Name":"Banana","ExpiryDate":"2019-07-26T00:00:00"}
```

`DateTime` and `DateTimeOffset` can also be deserialized with `JsonSerializer`:

C#

```
using System.Text.Json;

public class Example
{
    private class Product
    {
        public string? Name { get; set; }
        public DateTime ExpiryDate { get; set; }
    }

    public static void Main(string[] args)
    {
        string json = @"{""Name"":""Banana"" , ""ExpiryDate"":""2019-07-
26T00:00:00""}";
        Product p = JsonSerializer.Deserialize<Product>(json)!;
        Console.WriteLine(p.Name);
        Console.WriteLine(p.ExpiryDate);
    }
}

// The example displays output similar to the following:
// Banana
// 7/26/2019 12:00:00 AM
```

With default options, input `DateTime` and `DateTimeOffset` text representations must conform to the extended ISO 8601-1:2019 profile. Attempting to deserialize representations that don't conform to the profile will cause `JsonSerializer` to throw a `JsonException`:

C#

```
using System.Text.Json;

public class Example
{
    private class Product
    {
        public string? Name { get; set; }
        public DateTime ExpiryDate { get; set; }
    }

    public static void Main(string[] args)
    {
        string json = @"
{""Name"":""Banana"" , ""ExpiryDate"":""26/07/2019""}";
        try
        {
            Product _ = JsonSerializer.Deserialize<Product>(json)!;
        }
        catch (JsonException e)
```

```

        {
            Console.WriteLine(e.Message);
        }
    }

// The example displays the following output:
// The JSON value could not be converted to System.DateTime. Path:
$.ExpiryDate | LineNumber: 0 | BytePositionInLine: 42.

```

The `JsonDocument` provides structured access to the contents of a JSON payload, including `DateTime` and `DateTimeOffset` representations. The following example shows how to calculate the average temperature on Mondays from a collection of temperatures:

C#

```

using System.Text.Json;

public class Example
{
    private static double ComputeAverageTemperatures(string json)
    {
        JsonDocumentOptions options = new JsonDocumentOptions
        {
            AllowTrailingCommas = true
        };

        using (JsonDocument document = JsonDocument.Parse(json, options))
        {
            int sumOfAllTemperatures = 0;
            int count = 0;

            foreach (JsonElement element in
document.RootElement.EnumerateArray())
            {
                DateTimeOffset date =
element.GetProperty("date").GetDateTimeOffset();

                if (date.DayOfWeek == DayOfWeek.Monday)
                {
                    int temp = element.GetProperty("temp").GetInt32();
                    sumOfAllTemperatures += temp;
                    count++;
                }
            }

            double averageTemp = (double)sumOfAllTemperatures / count;
            return averageTemp;
        }
    }
}

```

```

public static void Main(string[] args)
{
    string json =
        @"[ " +
        @"{ " +
            @"""date"": """2013-01-07T00:00:00Z""", " +
            @"""temp"": 23, " +
        @"}, " +
        @"{ " +
            @"""date"": """2013-01-08T00:00:00Z""", " +
            @"""temp"": 28, " +
        @"}, " +
        @"{ " +
            @"""date"": """2013-01-14T00:00:00Z""", " +
            @"""temp"": 8, " +
        @"}, " +
        @"]";

    Console.WriteLine(ComputeAverageTemperatures(json));
}

// The example displays the following output:
// 15.5

```

Attempting to compute the average temperature given a payload with non-compliant `DateTime` representations will cause `JsonDocument` to throw a [FormatException](#):

C#

```

using System.Text.Json;

public class Example
{
    private static double ComputeAverageTemperatures(string json)
    {
        JsonDocumentOptions options = new JsonDocumentOptions
        {
            AllowTrailingCommas = true
        };

        using (JsonDocument document = JsonDocument.Parse(json, options))
        {
            int sumOfAllTemperatures = 0;
            int count = 0;

            foreach (JsonElement element in
document.RootElement.EnumerateArray())
            {
                DateTimeOffset date =
element.GetProperty("date").GetDateTimeOffset();

                if (date.DayOfWeek == DayOfWeek.Monday)

```

```

        {
            int temp = element.GetProperty("temp").GetInt32();
            sumOfAllTemperatures += temp;
            count++;
        }
    }

    double averageTemp = (double)sumOfAllTemperatures / count;
    return averageTemp;
}

public static void Main(string[] args)
{
    // Computing the average temperatures will fail because the
    DateTimeOffset
    // values in the payload do not conform to the extended ISO 8601-
    1:2019 profile.
    string json =
        @"[ " +
        @"{ " +
        @"""date"": ""2013/01/07 00:00:00Z""," +
        @"""temp"": 23," +
        @"}, " +
        @"{ " +
        @"""date"": ""2013/01/08 00:00:00Z""," +
        @"""temp"": 28," +
        @"}, " +
        @"{ " +
        @"""date"": ""2013/01/14 00:00:00Z""," +
        @"""temp"": 8," +
        @"}, " +
        @"]";
}

Console.WriteLine(ComputeAverageTemperatures(json));
}
}

// The example displays the following output:
// Unhandled exception.System.FormatException: One of the identified items
was in an invalid format.
//     at System.Text.Json.JsonElement.GetDateTimeOffset()

```

The lower level [Utf8JsonWriter](#) writes [DateTime](#) and [DateTimeOffset](#) data:

```

C#

using System.Text;
using System.Text.Json;

public class Example
{
    public static void Main(string[] args)

```

```

    {
        JsonWriterOptions options = new JsonWriterOptions
        {
            Indented = true
        };

        using (MemoryStream stream = new MemoryStream())
        {
            using (Utf8JsonWriter writer = new Utf8JsonWriter(stream,
options))
            {
                writer.WriteStartObject();
                writer.WriteString("date", DateTimeOffset.UtcNow);
                writer.WriteNumber("temp", 42);
                writer.WriteEndObject();
            }

            string json = Encoding.UTF8.GetString(stream.ToArray());
            Console.WriteLine(json);
        }
    }

// The example output similar to the following:
// {
//     "date": "2019-07-26T00:00:00+00:00",
//     "temp": 42
// }

```

[Utf8JsonReader](#) parses [DateTime](#) and [DateTimeOffset](#) data:

C#

```

using System.Text;
using System.Text.Json;

public class Example
{
    public static void Main(string[] args)
    {
        byte[] utf8Data = Encoding.UTF8.GetBytes(@"""2019-07-
26T00:00:00""");

        Utf8JsonReader json = new Utf8JsonReader(utf8Data);
        while (json.Read())
        {
            if (json.TokenType == JsonTokenType.String)
            {
                Console.WriteLine(json.TryGetDateTime(out DateTime
datetime));
                Console.WriteLine(datetime);
                Console.WriteLine(json.GetDateTime());
            }
        }
    }
}

```

```
        }
    }

// The example displays output similar to the following:
// True
// 7/26/2019 12:00:00 AM
// 7/26/2019 12:00:00 AM
```

Attempting to read non-compliant formats with [Utf8JsonReader](#) will cause it to throw a [FormatException](#):

C#

```
using System.Text;
using System.Text.Json;

public class Example
{
    public static void Main(string[] args)
    {
        byte[] utf8Data = Encoding.UTF8.GetBytes(@"""2019/07/26
00:00:00""");

        Utf8JsonReader json = new Utf8JsonReader(utf8Data);
        while (json.Read())
        {
            if (json.TokenType == JsonTokenType.String)
            {
                Console.WriteLine(json.TryGetDateTime(out DateTime
datetime));
                Console.WriteLine(datetime);

                DateTime _ = json.GetDateTime();
            }
        }
    }

// The example displays the following output:
// False
// 1/1/0001 12:00:00 AM
// Unhandled exception. System.FormatException: The JSON value is not in a
supported DateTime format.
//      at System.Text.Json.Utf8JsonReader.GetDateTime()
```

Serialize DateOnly and TimeOnly properties

With .NET 7+, `System.Text.Json` supports serializing and deserializing [DateOnly](#) and [TimeOnly](#) types. Consider the following object:

```
C#
```

```
sealed file record Appointment(
    Guid Id,
    string Description,
    DateOnly Date,
    TimeOnly StartTime,
    TimeOnly EndTime);
```

The following example serializes an `Appointment` object, displays the resulting JSON, and then deserializes it back into a new instance of the `Appointment` type. Finally, the original and newly deserialized instances are compared for equality and the results are written to the console:

```
C#
```

```
Appointment originalAppointment = new(
    Id: Guid.NewGuid(),
    Description: "Take dog to veterinarian.",
    Date: new DateOnly(2002, 1, 13),
    StartTime: new TimeOnly(5,15),
    EndTime: new TimeOnly(5, 45));
string serialized = JsonSerializer.Serialize(originalAppointment);

Console.WriteLine($"Resulting JSON: {serialized}");

Appointment deserializedAppointment =
    JsonSerializer.Deserialize<Appointment>(serialized)!;

bool valuesAreTheSame = originalAppointment == deserializedAppointment;
Console.WriteLine($"""
    Original record has the same values as the deserialized record:
{valuesAreTheSame}
""");
```

In the preceding code:

- An `Appointment` object is instantiated and assigned to the `appointment` variable.
- The `appointment` instance is serialized to JSON using `JsonSerializer.Serialize`.
- The resulting JSON is written to the console.
- The JSON is deserialized back into a new instance of the `Appointment` type using `JsonSerializer.Deserialize`.
- The original and newly deserialized instances are compared for equality.
- The result of the comparison is written to the console.

Custom support for `DateTime` and `DateTimeOffset`

When using `JsonSerializer`

If you want the serializer to perform custom parsing or formatting, you can implement [custom converters](#). Here are a few examples:

`DateTimeOffset.Parse` and `DateTimeOffset.ToString`

If you can't determine the formats of your input `DateTime` or `DateTimeOffset` text representations, you can use the `DateTimeOffset.Parse` method in your converter read logic. This method allows you to use .NET's extensive support for parsing various `DateTime` and `DateTimeOffset` text formats, including non-ISO 8601 strings and ISO 8601 formats that don't conform to the extended ISO 8601-1:2019 profile. This approach is less performant than using the serializer's native implementation.

For serializing, you can use the `DateTimeOffset.ToString` method in your converter write logic. This method allows you to write `DateTime` and `DateTimeOffset` values using any of the [standard date and time formats](#), and the [custom date and time formats](#). This approach is also less performant than using the serializer's native implementation.

C#

```
using System.Diagnostics;
using System.Text.Json;
using System.Text.Json.Serialization;
using System.Text.RegularExpressions;

namespace DateTimeConverterExamples;

public class DateTimeConverterUsingDateTimeParse : JsonConverter<DateTime>
{
    public override DateTime Read(ref Utf8JsonReader reader, Type typeToConvert, JsonSerializerOptions options)
    {
        Debug.Assert(typeToConvert == typeof(DateTime));
        return DateTime.Parse(reader.GetString() ?? string.Empty);
    }

    public override void Write(Utf8JsonWriter writer, DateTime value,
JsonSerializerOptions options)
    {
        writer.WriteStringValue(value.ToString());
    }
}
```

```
class Program
{
    private static void ParseDateTimeWithDefaultOptions()
    {
        DateTime _ = JsonSerializer.Deserialize<DateTime>(@"""04-10-2008
6:30 AM""");
    }

    private static void FormatDateTimeWithDefaultOptions()
    {
        Console.WriteLine(JsonSerializer.Serialize(DateTime.Parse("04-10-
2008 6:30 AM -4")));
    }

    private static void ProcessDateTimeWithCustomConverter()
    {
        JsonSerializerOptions options = new JsonSerializerOptions();
        options.Converters.Add(new DateTimeConverterUsingDateTimeParse());

        string testDateTimeStr = "04-10-2008 6:30 AM";
        string testDateTimeJson = @""" + testDateTimeStr + @"""";

        DateTime resultDateTime = JsonSerializer.Deserialize<DateTime>
(testDateTimeJson, options);
        Console.WriteLine(resultDateTime);

        string resultDateTimeJson =
JsonSerializer.Serialize(DateTime.Parse(testDateTimeStr), options);
        Console.WriteLine(Regex.Unescape(resultDateTimeJson));
    }

    static void Main(string[] args)
    {
        // Parsing non-compliant format as DateTime fails by default.
        try
        {
            ParseDateTimeWithDefaultOptions();
        }
        catch (JsonException e)
        {
            Console.WriteLine(e.Message);
        }

        // Formatting with default options prints according to extended ISO
        // 8601 profile.
        FormatDateTimeWithDefaultOptions();

        // Using converters gives you control over the serializers parsing
        // and formatting.
        ProcessDateTimeWithCustomConverter();
    }
}

// The example displays output similar to the following:
```

```
// The JSON value could not be converted to System.DateTime. Path: $ |  
LineNumber: 0 | BytePositionInLine: 20.  
// "2008-04-10T06:30:00-04:00"  
// 4/10/2008 6:30:00 AM  
// "4/10/2008 6:30:00 AM"
```

ⓘ Note

When implementing `JsonConverter<T>`, and `T` is `DateTime`, the `typeToConvert` parameter will always be `typeof(DateTime)`. The parameter is useful for handling polymorphic cases and when using generics to get `typeof(T)` in a performant way.

Utf8Parser and Utf8Formatter

You can use fast UTF-8-based parsing and formatting methods in your converter logic if your input `DateTime` or `DateTimeOffset` text representations are compliant with one of the "R", "I", "O", or "G" [standard date and time format strings](#), or you want to write according to one of these formats. This approach is much faster than using `S DateTime(Offset).Parse` and `DateTime(Offset).ToString`.

The following example shows a custom converter that serializes and deserializes `DateTime` values according to [the "R" standard format](#):

C#

```
using System.Buffers;  
using System.Buffers.Text;  
using System.Diagnostics;  
using System.Text.Json;  
using System.Text.Json.Serialization;  
  
namespace DateTimeConverterExamples;  
  
// This converter reads and writes DateTime values according to the "R"  
// standard format specifier:  
// https://learn.microsoft.com/dotnet/standard/base-types/standard-date-and-  
// time-format-strings#the-rfc1123-r-r-format-specifier.  
public class DateTimeConverterForCustomStandardFormatR :  
JsonConverter<DateTime>  
{  
    public override DateTime Read(ref Utf8JsonReader reader, Type  
typeToConvert, JsonSerializerOptions options)  
    {  
        Debug.Assert(typeToConvert == typeof(DateTime));  
  
        if (Utf8Parser.TryParse(reader.ValueSpan, out DateTime value, out _,  
'R'))
```

```

        {
            return value;
        }

        throw new FormatException();
    }

    public override void Write(Utf8JsonWriter writer, DateTime value,
JsonSerializerOptions options)
{
    // The "R" standard format will always be 29 bytes.
    Span<byte> utf8Date = new byte[29];

    bool result = Utf8Formatter.TryFormat(value, utf8Date, out _, new
StandardFormat('R'));
    Debug.Assert(result);

    writer.WriteStringValue(utf8Date);
}
}

class Program
{
    private static void ParseDateTimeWithDefaultOptions()
    {
        DateTime _ = JsonSerializer.Deserialize(@"
Thu, 25 Jul
2019 13:36:07 GMT");
    }

    private static void ProcessDateTimeWithCustomConverter()
    {
        JsonSerializerOptions options = new JsonSerializerOptions();
        options.Converters.Add(new
DateTimeConverterForCustomStandardFormatR());

        string testDateTimeStr = "Thu, 25 Jul 2019 13:36:07 GMT";
        string testDateTimeJson = @"" + testDateTimeStr + @_;

        DateTime resultDateTime = JsonSerializer.Deserialize

```

```
        {
            Console.WriteLine(e.Message);
        }

        // Using converters gives you control over the serializers parsing
        // and formatting.
        ProcessDateTimeWithCustomConverter();
    }
}

// The example displays output similar to the following:
// The JSON value could not be converted to System.DateTime.Path: $ | 
LineNumber: 0 | BytePositionInLine: 31.
// 7/25/2019 1:36:07 PM
// "Thu, 25 Jul 2019 09:36:07 GMT"
```

ⓘ Note

The "R" standard format will always be 29 characters long.

The "l" (lowercase "L") format isn't documented with the other **standard date and time format strings** because it's supported only by the `Utf8Parser` and `Utf8Formatter` types. The format is lowercase RFC 1123 (a lowercase version of the "R" format). For example, "thu, 25 jul 2019 06:36:07 gmt".

Use `DateTime(Offset).Parse` as a fallback

If you generally expect your input `DateTime` or `DateTimeOffset` data to conform to the extended ISO 8601-1:2019 profile, you can use the serializer's native parsing logic. You can also implement a fallback mechanism. The following example shows that, after failing to parse a `DateTime` text representation using `TryGetDateTime(DateTime)`, the converter successfully parses the data using `Parse(String)`:

C#

```
using System.Diagnostics;
using System.Text.Json;
using System.Text.Json.Serialization;
using System.Text.RegularExpressions;

namespace DateTimeConverterExamples;

public class DateTimeConverterUsingDateTimeParseAsFallback :
JsonConverter<DateTime>
{
    public override DateTime Read(ref Utf8JsonReader reader, Type
typeToConvert, JsonSerializerOptions options)
```

```
{  
    Debug.Assert(typeToConvert == typeof(DateTime));  
  
    if (!reader.TryGetDateTime(out DateTime value))  
    {  
        value = DateTime.Parse(reader.GetString()!);  
    }  
  
    return value;  
}  
  
public override void Write(Utf8JsonWriter writer, DateTime value,  
JsonSerializerOptions options)  
{  
    writer.WriteStringValue(value.ToString("dd/MM/yyyy"));  
}  
}  
  
class Program  
{  
    private static void ParseDateTimeWithDefaultOptions()  
    {  
        DateTime _ = JsonSerializer.Deserialize<DateTime>(@"""2019-07-16  
16:45:27.4937872+00:00""");  
    }  
  
    private static void ProcessDateTimeWithCustomConverter()  
    {  
        JsonSerializerOptions options = new JsonSerializerOptions();  
        options.Converters.Add(new  
DateTimeConverterUsingDateTimeParseAsFallback());  
  
        string testDateTimeStr = "2019-07-16 16:45:27.4937872+00:00";  
        string testDateTimeJson = @""" + testDateTimeStr + @""";  
  
        DateTime resultDateTime = JsonSerializer.Deserialize<DateTime>  
(testDateTimeJson, options);  
        Console.WriteLine(resultDateTime);  
  
        string resultDateTimeJson =  
JsonSerializer.Serialize(DateTime.Parse(testDateTimeStr), options);  
        Console.WriteLine(Regex.Unescape(resultDateTimeJson));  
    }  
  
    static void Main(string[] args)  
    {  
        // Parsing non-compliant format as DateTime fails by default.  
        try  
        {  
            ParseDateTimeWithDefaultOptions();  
        }  
        catch (JsonException e)  
        {  
            Console.WriteLine(e.Message);  
        }  
    }  
}
```

```

    // Using converters gives you control over the serializers parsing
    // and formatting.
    ProcessDateTimeWithCustomConverter();
}
}

// The example displays output similar to the following:
// The JSON value could not be converted to System.DateTime.Path: $ |
// LineNumber: 0 | BytePositionInLine: 35.
// 7/16/2019 4:45:27 PM
// "16/07/2019"

```

Use Unix epoch date format

The following converters handle Unix epoch format with or without a time zone offset (values such as `/Date(1590863400000-0700)/` or `/Date(1590863400000)/`):

C#

```

sealed class UnixEpochDateTimeOffsetConverter :
JsonConverter<DateTimeOffset>
{
    static readonly DateTimeOffset s_epoch = new DateTimeOffset(1970, 1, 1,
0, 0, 0, TimeSpan.Zero);
    static readonly Regex s_regex = new Regex("^/Date\\\"(([+-]*)((\\d{2})\\(\\d{2}\\))\\)\\/$", RegexOptions.CultureInvariant);

    public override DateTimeOffset Read(ref Utf8JsonReader reader, Type
typeToConvert, JsonSerializerOptions options)
    {
        string formatted = reader.GetString()!;
        Match match = s_regex.Match(formatted);

        if (
            !match.Success
            || !long.TryParse(match.Groups[1].Value,
System.Globalization.NumberStyles.Integer, CultureInfo.InvariantCulture, out
long unixTime)
            || !int.TryParse(match.Groups[3].Value,
System.Globalization.NumberStyles.Integer, CultureInfo.InvariantCulture, out
int hours)
            || !int.TryParse(match.Groups[4].Value,
System.Globalization.NumberStyles.Integer, CultureInfo.InvariantCulture, out
int minutes))
        {
            throw new JsonException();
        }

        int sign = match.Groups[2].Value[0] == '+' ? 1 : -1;
        TimeSpan utcOffset = new TimeSpan(hours * sign, minutes * sign, 0);
    }
}

```

```

        return s_epoch.AddMilliseconds(unixTime).ToOffset(utcOffset);
    }

    public override void Write(Utf8JsonWriter writer, DateTimeOffset value,
JsonSerializerOptions options)
{
    long unixTime = Convert.ToInt64((value -
s_epoch).TotalMilliseconds);
    TimeSpan utcOffset = value.Offset;

    string formatted = string.Create(CultureInfo.InvariantCulture,
$""/Date({unixTime}{{(utcOffset >= TimeSpan.Zero ? "+" : "-")}{utcOffset:hhmm}})/");
    writer.WriteStringValue(formatted);
}
}

```

C#

```

sealed class UnixEpochDateTimeConverter : JsonConverter<DateTime>
{
    static readonly DateTime s_epoch = new DateTime(1970, 1, 1, 0, 0, 0);
    static readonly Regex s_regex = new Regex("^\u002fDate\\u0024(([-]+)*\\d+)\\u0024", RegexOptions.CultureInvariant);

    public override DateTime Read(ref Utf8JsonReader reader, Type
typeToConvert, JsonSerializerOptions options)
    {
        string formatted = reader.GetString()!;
        Match match = s_regex.Match(formatted);

        if (
            !match.Success
            || !long.TryParse(match.Groups[1].Value,
System.Globalization.NumberStyles.Integer, CultureInfo.InvariantCulture, out
long unixTime))
        {
            throw new JsonException();
        }

        return s_epoch.AddMilliseconds(unixTime);
    }

    public override void Write(Utf8JsonWriter writer, DateTimeOffset value,
JsonSerializerOptions options)
    {
        long unixTime = Convert.ToInt64((value -
s_epoch).TotalMilliseconds);

        string formatted = string.Create(CultureInfo.InvariantCulture,
$""/Date({unixTime})/");
    }
}

```

```
        writer.WriteStringValue(formatted);
    }
}
```

When using [Utf8JsonWriter](#)

If you want to write a custom [DateTime](#) or [DateTimeOffset](#) text representation with [Utf8JsonWriter](#), you can format your custom representation to a [String](#), [ReadOnlySpan<Byte>](#), [ReadOnlySpan<Char>](#), or [JsonEncodedText](#), then pass it to the corresponding [Utf8JsonWriter.WriteStringValue](#) or [Utf8JsonWriter.WriteString](#) method.

The following example shows how a custom [DateTime](#) format can be created with [ToString\(String, IFormatProvider\)](#) and then written with the [WriteStringValue\(String\)](#) method:

C#

```
using System.Globalization;
using System.Text;
using System.Text.Json;

public class Example
{
    public static void Main(string[] args)
    {
        var options = new JsonWriterOptions
        {
            Indented = true
        };

        using (var stream = new MemoryStream())
        {
            using (var writer = new Utf8JsonWriter(stream, options))
            {
                string dateStr = DateTime.UtcNow.ToString("F",
CultureInfo.InvariantCulture);

                writer.WriteStartObject();
                writer.WriteString("date", dateStr);
                writer.WriteNumber("temp", 42);
                writer.WriteEndObject();
            }

            string json = Encoding.UTF8.GetString(stream.ToArray());
            Console.WriteLine(json);
        }
    }
}

// The example displays output similar to the following:
```

```
// {
//     "date": "Tuesday, 27 August 2019 19:21:44",
//     "temp": 42
// }
```

When using [Utf8JsonReader](#)

If you want to read a custom [DateTime](#) or [DateTimeOffset](#) text representation with [Utf8JsonReader](#), you can get the value of the current JSON token as a [String](#) using [GetString\(\)](#) method, then parse the value using custom logic.

The following example shows how a custom [DateTimeOffset](#) text representation can be retrieved using the [GetString\(\)](#) method, then parsed using [ParseExact\(String, String, IFormatProvider\)](#):

C#

```
using System.Globalization;
using System.Text;
using System.Text.Json;

public class Example
{
    public static void Main(string[] args)
    {
        byte[] utf8Data = Encoding.UTF8.GetBytes(@"""Friday, 26 July 2019
00:00:00""");

        var json = new Utf8JsonReader(utf8Data);
        while (json.Read())
        {
            if (json.TokenType == JsonTokenType.String)
            {
                string value = json.GetString();
                DateTimeOffset dto = DateTimeOffset.ParseExact(value, "F",
CultureInfo.InvariantCulture);
                Console.WriteLine(dto);
            }
        }
    }
}

// The example displays output similar to the following:
// 7/26/2019 12:00:00 AM -04:00
```

The extended ISO 8601-1:2019 profile in [System.Text.Json](#)

Date and time components

The extended ISO 8601-1:2019 profile implemented in [System.Text.Json](#) defines the following components for date and time representations. These components are used to define various supported levels of granularity when parsing and formatting [DateTime](#) and [DateTimeOffset](#) representations.

| Component | Format | Description |
|-----------------|-------------------------|--|
| Year | "yyyy" | 0001-9999 |
| Month | "MM" | 01-12 |
| Day | "dd" | 01-28, 01-29, 01-30, 01-31 based on month/year. |
| Hour | "HH" | 00-23 |
| Minute | "mm" | 00-59 |
| Second | "ss" | 00-59 |
| Second fraction | "FFFFFF" | Minimum of one digit, maximum of 16 digits. |
| Time offset | "K" | Either "Z" or "(+/-)HH:mm". |
| Partial time | "HH:mm:ss[FFFFFF]" | Time without UTC offset information. |
| Full date | "yyyy'-'MM'-'dd" | Calendar date. |
| Full time | "Partial time'K" | UTC of day or Local time of day with the time offset between local time and UTC. |
| Date time | "Full date"T"Full time" | Calendar date and time of day, for example, 2019-07-26T16:59:57-05:00. |

Support for parsing

The following levels of granularity are defined for parsing:

1. 'Full date'
 - a. "yyyy'-'MM'-'dd"
2. "'Full date"T"Hour":"Minute"
 - a. "yyyy'-'MM'-'dd'T'HH':'mm"
3. "'Full date"T"Partial time"
 - a. "yyyy'-'MM'-'dd'T'HH':'mm':'ss" ([The Sortable \("s"\) Format Specifier](#))

- b. "yyyy'-'MM'-'dd'T'HH':'mm':'ss'.FFFFFFFFFF"
4. ""Full date" "T" "Time hour" ":" "Minute" "Time offset""
- a. "yyyy'-'MM'-'dd'T'HH':'mmZ"
 - b. "yyyy'-'MM'-'dd'T'HH':'mm('+'/-')HH':'mm"
5. 'Date time'
- a. "yyyy'-'MM'-'dd'T'HH':'mm':'ssZ"
 - b. "yyyy'-'MM'-'dd'T'HH':'mm':'ss'.FFFFFFFFFFZ"
 - c. "yyyy'-'MM'-'dd'T'HH':'mm':'ss('+'/-')HH':'mm"
 - d. "yyyy'-'MM'-'dd'T'HH':'mm':'ss'.FFFFFFFFFF('+'/-')HH':'mm"

This level of granularity is compliant with [RFC 3339](#), a widely adopted profile of ISO 8601 used for interchanging date and time information. However, there are a few restrictions in the `System.Text.Json` implementation.

- RFC 3339 doesn't specify a maximum number of fractional-second digits, but specifies that at least one digit must follow the period, if a fractional-second section is present. The implementation in `System.Text.Json` allows up to 16 digits (to support interop with other programming languages and frameworks), but parses only the first seven. A `JsonException` will be thrown if there are more than 16 fractional second digits when reading `DateTime` and `DateTimeOffset` instances.
- RFC 3339 allows the "T" and "Z" characters to be "t" or "z" respectively, but allows applications to limit support to just the upper-case variants. The implementation in `System.Text.Json` requires them to be "T" and "Z". A `JsonException` will be thrown if input payloads contain "t" or "z" when reading `DateTime` and `DateTimeOffset` instances.
- RFC 3339 specifies that the date and time sections are separated by "T", but allows applications to separate them by a space (" ") instead. `System.Text.Json` requires date and time sections to be separated with "T". A `JsonException` will be thrown if input payloads contain a space (" ") when reading `DateTime` and `DateTimeOffset` instances.

If there are decimal fractions for seconds, there must be at least one digit. `2019-07-26T00:00:00.` isn't allowed. While up to 16 fractional digits are allowed, only the first seven are parsed. Anything beyond that is considered a zero. For example, `2019-07-26T00:00:00.1234567890` will be parsed as if it's `2019-07-26T00:00:00.1234567`. This approach maintains compatibility with the `DateTime` implementation, which is limited to this resolution.

Leap seconds aren't supported.

Support for formatting

The following levels of granularity are defined for formatting:

1. "Full date"|"Partial time"

a. "yyyy'-'MM'-'dd'T'HH':'mm':'ss" ([The Sortable \("s"\) Format Specifier](#))

Used to format a [DateTime](#) without fractional seconds and without offset information.

b. "yyyy'-'MM'-'dd'T'HH':'mm':'ss'.FFFFFF"

Used to format a [DateTime](#) with fractional seconds but without offset information.

2. 'Date time'

a. "yyyy'-'MM'-'dd'T'HH':'mm':'ssZ"

Used to format a [DateTime](#) without fractional seconds but with a UTC offset.

b. "yyyy'-'MM'-'dd'T'HH':'mm':'ss'.FFFFFFZ"

Used to format a [DateTime](#) with fractional seconds and with a UTC offset.

c. "yyyy'-'MM'-'dd'T'HH':'mm':'ss('+'/-')HH':'mm"

Used to format a [DateTime](#) or [DateTimeOffset](#) without fractional seconds but with a local offset.

d. "yyyy'-'MM'-'dd'T'HH':'mm':'ss'.FFFFFF('+'/-')HH':'mm"

Used to format a [DateTime](#) or [DateTimeOffset](#) with fractional seconds and with a local offset.

This level of granularity is compliant with [RFC 3339](#).

If the [round-trip format](#) representation of a [DateTime](#) or [DateTimeOffset](#) instance has trailing zeros in its fractional seconds, then [JsonSerializer](#) and [Utf8JsonWriter](#) will format a representation of the instance without trailing zeros. For example, a [DateTime](#) instance whose [round-trip format](#) representation is `2019-04-24T14:50:17.101000Z`, will be formatted as `2019-04-24T14:50:17.101Z` by [JsonSerializer](#) and [Utf8JsonWriter](#).

If the [round-trip format](#) representation of a [DateTime](#) or [DateTimeOffset](#) instance has all zeros in its fractional seconds, then [JsonSerializer](#) and [Utf8JsonWriter](#) will format a representation of the instance without fractional seconds. For example, a [DateTime](#)

instance whose [round-trip format](#) representation is `2019-04-24T14:50:17.0000000+02:00`, will be formatted as `2019-04-24T14:50:17+02:00` by [JsonSerializer](#) and [Utf8JsonWriter](#).

Truncating zeros in fractional-second digits allows the smallest output needed to preserve information on a round trip to be written.

A maximum of seven fractional-second digits are written. This maximum aligns with the [DateTime](#) implementation, which is limited to this resolution.

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

Time zone overview

Article • 02/15/2023

The [TimeZoneInfo](#) class simplifies the creation of time zone-aware applications. The [TimeZone](#) class supports working with the local time zone and Coordinated Universal Time (UTC). The [TimeZoneInfo](#) class supports both of these zones as well as any time zone about which information is predefined in the registry. You can also use [TimeZoneInfo](#) to define custom time zones that the system has no information about.

Time zone essentials

A time zone is a geographical region in which the same time is used. Typically, but not always, adjacent time zones are one hour apart. The time in any of the world's time zones can be expressed as an offset from Coordinated Universal Time (UTC).

Many of the world's time zones support daylight saving time. Daylight saving time tries to maximize daylight hours by advancing the time forward by one hour in the spring or early summer, and returning to the normal (or standard) time in the late summer or fall. These changes to and from standard time are known as adjustment rules.

The transition to and from daylight saving time in a particular time zone can be defined either by a fixed or a floating adjustment rule. A fixed adjustment rule sets a particular date on which the transition to or from daylight saving time occurs each year. For example, a transition from daylight saving time to standard time that occurs each year on October 25 follows a fixed adjustment rule. Much more common are floating adjustment rules, which set a particular day of a particular week of a particular month for the transition to or from daylight saving time. For example, a transition from standard time to daylight saving time that occurs on the third Sunday of March follows a floating adjustment rule.

For time zones that support adjustment rules, the transition to and from daylight saving time creates two kinds of anomalous times: invalid times and ambiguous times. An invalid time is a nonexistent time created by the transition from standard time to daylight saving time. For example, if this transition occurs on a particular day at 2:00 A.M. and causes the time to change to 3:00 A.M., each time interval between 2:00 A.M. and 2:59:59 A.M. is invalid. An ambiguous time is a time that can be mapped to two different times in a single time zone. It is created by the transition from daylight saving time to standard time. For example, if this transition occurs on a particular day at 2:00 A.M. and causes the time to change to 1:00 A.M., each time interval between 1:00 A.M. and 1:59:59 A.M. can be interpreted as either a standard time or a daylight saving time.

Time zone terminology

The following table defines terms commonly used when working with time zones and developing time zone-aware applications.

| Term | Definition |
|-----------------|---|
| Adjustment rule | A rule that defines when the transition from standard time to daylight saving time and back from daylight saving time to standard time occurs. Each adjustment rule has a start and end date that defines when the rule is in place (for example, the adjustment rule is in place from January 1, 1986, to December 31, 2006), a delta (the amount of time by which the standard time changes as a result of the application of the adjustment rule), and information about the specific date and time that the transitions are to occur during the adjustment period. Transitions can follow either a fixed rule or a floating rule. |
| Ambiguous time | A time that can be mapped to two different times in a single time zone. It occurs when the clock time is adjusted back in time, such as during the transition from a time zone's daylight saving time to its standard time. For example, if this transition occurs on a particular day at 2:00 A.M. and causes the time to change to 1:00 A.M., each time interval between 1:00 A.M. and 1:59:59 A.M. can be interpreted as either a standard time or a daylight saving time. |
| Fixed rule | An adjustment rule that sets a particular date for the transition to or from daylight saving time. For example, a transition from daylight saving time to standard time that occurs each year on October 25 follows a fixed adjustment rule. |
| Floating rule | An adjustment rule that sets a particular day of a particular week of a particular month for the transition to or from daylight saving time. For example, a transition from standard time to daylight saving time that occurs on the third Sunday of March follows a floating adjustment rule. |
| Invalid time | A nonexistent time that is an artifact of the transition from standard time to daylight saving time. It occurs when the clock time is adjusted forward in time, such as during the transition from a time zone's standard time to its daylight saving time. For example, if this transition occurs on a particular day at 2:00 A.M. and causes the time to change to 3:00 A.M., each time interval between 2:00 A.M. and 2:59:59 A.M. is invalid. |
| Transition time | Information about a specific time change, such as the change from daylight saving time to standard time or vice versa, in a particular time zone. |

Time zones and the `TimeZoneInfo` class

In .NET, a `TimeZoneInfo` object represents a time zone. The `TimeZoneInfo` class includes a `GetAdjustmentRules` method that returns an array of `TimeZoneInfo.AdjustmentRule` objects. Each element of this array provides information about the transition to and

from daylight saving time for a particular time period. (For time zones that do not support daylight saving time, the method returns an empty array.) Each [TimeZoneInfo.AdjustmentRule](#) object has a [DaylightTransitionStart](#) and a [DaylightTransitionEnd](#) property that defines the particular date and time of the transition to and from daylight saving time. The [IsFixedDateRule](#) property indicates whether that transition is fixed or floating.

.NET relies on time zone information provided by the Windows operating system and stored in the registry. Due to the number of the earth's time zones, not all existing time zones are represented in the registry. In addition, because the registry is a dynamic structure, predefined time zones can be added to or removed from it. Finally, the registry does not necessarily contain historic time zone data. For example, in Windows XP the registry contains data about only a single set of time zone adjustments. Windows Vista supports dynamic time zone data, which means that a single time zone can have multiple adjustment rules that apply to specific intervals of years. However, most time zones that are defined in the Windows Vista registry and support daylight saving time have only one or two predefined adjustment rules.

The dependence of the [TimeZoneInfo](#) class on the registry means that a time zone-aware application cannot be certain that a particular time zone is defined in the registry. As a result, the attempt to instantiate a specific time zone (other than the local time zone or the time zone that represents UTC) should use exception handling. It should also provide some method of letting the application to continue if a required [TimeZoneInfo](#) object cannot be instantiated from the registry.

To handle the absence of a required time zone, the [TimeZoneInfo](#) class includes a [CreateCustomTimeZone](#) method, which you can use to create custom time zones that are not found in the registry. For details on creating a custom time zone, see [How to: Create time zones without adjustment rules](#) and [How to: Create time zones with adjustment rules](#). In addition, you can use the [ToSerializedString](#) method to convert a newly created time zone to a string and save it in a data store (such as a database, a text file, the registry, or an application resource). You can then use the [FromSerializedString](#) method to convert this string back to a [TimeZoneInfo](#) object. For details, see [How to: Save time zones to an embedded resource](#) and [How to: Restore time zones from an embedded resource](#).

Because each time zone is characterized by a base offset from UTC, as well as by an offset from UTC that reflects any existing adjustment rules, a time in one time zone can be easily converted to the time in another time zone. For this purpose, the [TimeZoneInfo](#) object includes several conversion methods, including:

- [ConvertTimeFromUtc](#), which converts UTC to the time in a designated time zone.

- [ConvertTimeToUtc](#), which converts the time in a designated time zone to UTC.
- [ConvertTime](#), which converts the time in one designated time zone to the time in another designated time zone.
- [ConvertTimeBySystemTimeZoneId](#), which uses time zone identifiers (instead of [TimeZoneInfo](#) objects) as parameters to convert the time in one designated time zone to the time in another designated time zone.

For details on converting times between time zones, see [Converting times between time zones](#).

See also

- [Dates, times, and time zones](#)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



.NET feedback

The .NET documentation is open source. Provide feedback [here](#).

 [Open a documentation issue](#)

 [Provide product feedback](#)

How to: Use time zones in date and time arithmetic

Article • 09/15/2021

Ordinarily, when you perform date and time arithmetic using [DateTime](#) or [DateTimeOffset](#) values, the result does not reflect any time zone adjustment rules. This is true even when the time zone of the date and time value is clearly identifiable (for example, when the [Kind](#) property is set to [Local](#)). This topic shows how to perform arithmetic operations on date and time values that belong to a particular time zone. The results of the arithmetic operations will reflect the time zone's adjustment rules.

To apply adjustment rules to date and time arithmetic

1. Implement some method of closely coupling a date and time value with the time zone to which it belongs. For example, declare a structure that includes both the date and time value and its time zone. The following example uses this approach to link a [DateTime](#) value with its time zone.

C#

```
// Define a structure for DateTime values for internal use only
internal struct TimeWithTimeZone
{
    TimeZoneInfo TimeZone;
    DateTime Time;
}
```

2. Convert a time to Coordinated Universal Time (UTC) by calling either the [ConvertTimeToUtc](#) method or the [ConvertTime](#) method.
3. Perform the arithmetic operation on the UTC time.
4. Convert the time from UTC to the original time's associated time zone by calling the [TimeZoneInfo.ConvertTime\(DateTime, TimeZoneInfo\)](#) method.

Example

The following example adds two hours and thirty minutes to March 9, 2008, at 1:30 A.M. Central Standard Time. The time zone's transition to daylight saving time occurs thirty minutes later, at 2:00 A.M. on March 9, 2008. Because the example follows the four steps

listed in the previous section, it correctly reports the resulting time as 5:00 A.M. on March 9, 2008.

C#

```
using System;

public struct TimeZoneTime
{
    public TimeZoneInfo TimeZone;
    public DateTime Time;

    public TimeZoneTime(TimeZoneInfo tz, DateTime time)
    {
        if (tz == null)
            throw new ArgumentNullException("The time zone cannot be a null reference.");
        
        this.TimeZone = tz;
        this.Time = time;
    }

    public TimeZoneTime AddTime(TimeSpan interval)
    {
        // Convert time to UTC
        DateTime utcTime = TimeZoneInfo.ConvertTimeToUtc(this.Time,
this.TimeZone);
        // Add time interval to time
        utcTime = utcTime.Add(interval);
        // Convert time back to time in time zone
        return new TimeZoneTime(this.TimeZone,
TimeZoneInfo.ConvertTime(utcTime,
TimeZoneInfo.Utc, this.TimeZone));
    }
}

public class TimeArithmetic
{
    public const string tzName = "Central Standard Time";

    public static void Main()
    {
        try
        {
            TimeZoneTime cstTime1, cstTime2;

            TimeZoneInfo cst = TimeZoneInfo.FindSystemTimeZoneById(tzName);
            DateTime time1 = new DateTime(2008, 3, 9, 1, 30, 0);
            TimeSpan twoAndAHalfHours = new TimeSpan(2, 30, 0);

            cstTime1 = new TimeZoneTime(cst, time1);
            cstTime2 = cstTime1.AddTime(twoAndAHalfHours);
            Console.WriteLine("{0} + {1} hours = {2}", cstTime1.Time,
```

```

        twoAndAHalfHours.ToString(),
        cstTime2.Time);
    }
    catch
    {
        Console.WriteLine("Unable to find {0}.", tzName);
    }
}

```

Both `DateTime` and `DateTimeOffset` values are disassociated from any time zone to which they might belong. To perform date and time arithmetic in a way that automatically applies a time zone's adjustment rules, the time zone to which any date and time value belongs must be immediately identifiable. This means that a date and time and its associated time zone must be tightly coupled. There are several ways to do this, which include the following:

- Assume that all times used in an application belong to a particular time zone. Although appropriate in some cases, this approach offers limited flexibility and possibly limited portability.
- Define a type that tightly couples a date and time with its associated time zone by including both as fields of the type. This approach is used in the code example, which defines a structure to store the date and time and the time zone in two member fields.

The example illustrates how to perform arithmetic operations on `DateTime` values so that time zone adjustment rules are applied to the result. However, `DateTimeOffset` values can be used just as easily. The following example illustrates how the code in the original example might be adapted to use `DateTimeOffset` instead of `DateTime` values.

C#

```

using System;

public struct TimeZoneTime
{
    public TimeZoneInfo TimeZone;
    public DateTimeOffset Time;

    public TimeZoneTime(TimeZoneInfo tz, DateTimeOffset time)
    {
        if (tz == null)
            throw new ArgumentNullException("The time zone cannot be a null
reference.");
        this.TimeZone = tz;
        this.Time = time;
    }
}

```

```

    }

    public TimeZoneTime AddTime(TimeSpan interval)
    {
        // Convert time to UTC
        DateTimeOffset utcTime = TimeZoneInfo.ConvertTime(this.Time,
TimeZoneInfo.Utc);
        // Add time interval to time
        utcTime = utcTime.Add(interval);
        // Convert time back to time in time zone
        return new TimeZoneTime(this.TimeZone,
TimeZoneInfo.ConvertTime(utcTime, this.TimeZone));
    }
}

public class TimeArithmetic
{
    public const string tzName = "Central Standard Time";

    public static void Main()
    {
        try
        {
            TimeZoneTime cstTime1, cstTime2;

            TimeZoneInfo cst = TimeZoneInfo.FindSystemTimeZoneById(tzName);
            DateTime time1 = new DateTime(2008, 3, 9, 1, 30, 0);
            TimeSpan twoAndAHalfHours = new TimeSpan(2, 30, 0);

            cstTime1 = new TimeZoneTime(cst,
                new DateTimeOffset(time1, cst.GetUtcOffset(time1)));
            cstTime2 = cstTime1.AddTime(twoAndAHalfHours);
            Console.WriteLine("{0} + {1} hours = {2}", cstTime1.Time,
twoAndAHalfHours.ToString(),
cstTime2.Time);
        }
        catch
        {
            Console.WriteLine("Unable to find {0}.", tzName);
        }
    }
}

```

Note that if this addition is simply performed on the `DateTimeOffset` value without first converting it to UTC, the result reflects the correct point in time but its offset does not reflect that of the designated time zone for that time.

Compiling the code

This example requires:

- That the `System` namespace be imported with the `using` statement (required in C# code).

See also

- [Dates, times, and time zones](#)
- [Performing arithmetic operations with dates and times](#)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

Converting between DateTime and DateTimeOffset

Article • 10/04/2022

Although the [DateTimeOffset](#) structure provides a greater degree of time zone awareness than the [DateTime](#) structure, [DateTime](#) parameters are used more commonly in method calls. Because of this approach, the ability to convert [DateTimeOffset](#) values to [DateTime](#) values and vice versa is important. This article shows how to perform these conversions in a way that preserves as much time zone information as possible.

ⓘ Note

Both the [DateTime](#) and the [DateTimeOffset](#) types have some limitations when representing times in time zones. With its [Kind](#) property, [DateTime](#) is able to reflect only Coordinated Universal Time (UTC) and the system's local time zone. [DateTimeOffset](#) reflects a time's offset from UTC, but it doesn't reflect the actual time zone to which that offset belongs. For more information about time values and support for time zones, see [Choosing Between DateTime, DateTimeOffset, TimeSpan, and TimeZoneInfo](#).

Conversions from DateTime to DateTimeOffset

The [DateTimeOffset](#) structure provides two equivalent ways to perform [DateTime](#) to [DateTimeOffset](#) conversion that are suitable for most conversions:

- The [DateTimeOffset](#) constructor, which creates a new [DateTimeOffset](#) object based on a [DateTime](#) value.
- The implicit conversion operator, which allows you to assign a [DateTime](#) value to a [DateTimeOffset](#) object.

For UTC and local [DateTime](#) values, the [Offset](#) property of the resulting [DateTimeOffset](#) value accurately reflects the UTC or local time zone offset. For example, the following code converts a UTC time to its equivalent [DateTimeOffset](#) value:

C#

```
DateTime utcTime1 = new DateTime(2008, 6, 19, 7, 0, 0);
utcTime1 = DateTime.SpecifyKind(utcTime1, DateTimeKind.Utc);
DateTimeOffset utcTime2 = utcTime1;
```

```
Console.WriteLine("Converted {0} {1} to a DateTimeOffset value of {2}",
    utcTime1,
    utcTime1.Kind,
    utcTime2);
// This example displays the following output to the console:
//     Converted 6/19/2008 7:00:00 AM Utc to a DateTimeOffset value of
6/19/2008 7:00:00 AM +00:00
```

In this case, the offset of the `utcTime2` variable is 00:00. Similarly, the following code converts a local time to its equivalent [DateTimeOffset](#) value:

```
C#
DateTime localTime1 = new DateTime(2008, 6, 19, 7, 0, 0);
localTime1 = DateTime.SpecifyKind(localTime1, DateTimeKind.Local);
DateTimeOffset localTime2 = localTime1;
Console.WriteLine("Converted {0} {1} to a DateTimeOffset value of {2}",
    localTime1,
    localTime1.Kind,
    localTime2);
// This example displays the following output to the console:
//     Converted 6/19/2008 7:00:00 AM Local to a DateTimeOffset value of
6/19/2008 7:00:00 AM -07:00
```

However, for [DateTime](#) values whose [Kind](#) property is [DateTimeKind.Unspecified](#), these two conversion methods produce a [DateTimeOffset](#) value whose offset is that of the local time zone. The conversion is shown in the following example, which is run in the U.S. Pacific Standard Time zone:

```
C#
DateTime time1 = new DateTime(2008, 6, 19, 7, 0, 0); // Kind is
DateTimeKind.Unspecified
DateTimeOffset time2 = time1;
Console.WriteLine("Converted {0} {1} to a DateTimeOffset value of {2}",
    time1,
    time1.Kind,
    time2);
// This example displays the following output to the console:
//     Converted 6/19/2008 7:00:00 AM Unspecified to a DateTimeOffset value
of 6/19/2008 7:00:00 AM -07:00
```

If the [DateTime](#) value reflects the date and time in something other than the local time zone or UTC, you can convert it to a [DateTimeOffset](#) value and preserve its time zone information by calling the overloaded [DateTimeOffset](#) constructor. For example, the following example instantiates a [DateTimeOffset](#) object that reflects Central Standard Time:

C#

```
DateTime time1 = new DateTime(2008, 6, 19, 7, 0, 0);      // Kind is
DateTimeKind.Unspecified
try
{
    DateTimeOffset time2 = new DateTimeOffset(time1,
                                                TimeZoneInfo.FindSystemTimeZoneById("Central Standard
Time").GetUtcOffset(time1));
    Console.WriteLine("Converted {0} {1} to a DateTime value of {2}",
                      time1,
                      time1.Kind,
                      time2);
}
// Handle exception if time zone is not defined in registry
catch (TimeZoneNotFoundException)
{
    Console.WriteLine("Unable to identify target time zone for conversion.");
}
// This example displays the following output to the console:
//   Converted 6/19/2008 7:00:00 AM Unspecified to a DateTime value of
6/19/2008 7:00:00 AM -05:00
```

The second parameter to this constructor overload is a [TimeSpan](#) object that represents the time's offset from UTC. Retrieve it by calling the [TimeZoneInfo.GetUtcOffset\(DateTime\)](#) method of the time's corresponding time zone. The method's single parameter is the [DateTime](#) value that represents the date and time to be converted. If the time zone supports daylight saving time, this parameter allows the method to determine the appropriate offset for that particular date and time.

Conversions from DateTimeOffset to DateTime

The [DateTime](#) property is most commonly used to perform [DateTimeOffset](#) to [DateTime](#) conversion. However, it returns a [DateTime](#) value whose [Kind](#) property is [Unspecified](#), as the following example illustrates:

C#

```
DateTime baseTime = new DateTime(2008, 6, 19, 7, 0, 0);
DateTimeOffset sourceTime;
DateTime targetTime;

// Convert UTC to DateTime value
sourceTime = new DateTimeOffset(baseTime, TimeSpan.Zero);
targetTime = sourceTime.DateTime;
Console.WriteLine("{0} converts to {1} {2}",
                  sourceTime,
                  targetTime,
```

```

        targetTime.Kind);

// Convert local time to DateTime value
sourceTime = new DateTimeOffset(baseTime,
                                TimeZoneInfo.Local.GetUtcOffset(baseTime));
targetTime = sourceTime.DateTime;
Console.WriteLine("{0} converts to {1} {2}",
                 sourceTime,
                 targetTime,
                 targetTime.Kind);

// Convert Central Standard Time to a DateTime value
try
{
    TimeSpan offset = TimeZoneInfo.FindSystemTimeZoneById("Central Standard
Time").GetUtcOffset(baseTime);
    sourceTime = new DateTimeOffset(baseTime, offset);
    targetTime = sourceTime.DateTime;
    Console.WriteLine("{0} converts to {1} {2}",
                     sourceTime,
                     targetTime,
                     targetTime.Kind);
}
catch (TimeZoneNotFoundException)
{
    Console.WriteLine("Unable to create DateTimeOffset based on U.S. Central
Standard Time.");
}
// This example displays the following output to the console:
//   6/19/2008 7:00:00 AM +00:00 converts to 6/19/2008 7:00:00 AM
Unspecified
//   6/19/2008 7:00:00 AM -07:00 converts to 6/19/2008 7:00:00 AM
Unspecified
//   6/19/2008 7:00:00 AM -05:00 converts to 6/19/2008 7:00:00 AM
Unspecified

```

The preceding example shows that any information about the [DateTimeOffset](#) value's relationship to UTC is lost by the conversion when the [DateTime](#) property is used. This behavior also affects [DateTimeOffset](#) values that correspond to UTC time or the system's local time because the [DateTime](#) structure reflects only those two time zones in its [Kind](#) property.

To preserve as much time zone information as possible when converting a [DateTimeOffset](#) to a [DateTime](#) value, you can use the [DateTimeOffset.UtcDateTime](#) and [DateTimeOffset.LocalDateTime](#) properties.

Converting a UTC time

To indicate that a converted [DateTime](#) value is the UTC time, you can retrieve the value of the [DateTimeOffset.UtcDateTime](#) property. It differs from the [DateTime](#) property in two ways:

- It returns a [DateTime](#) value whose [Kind](#) property is [Utc](#).
- If the [Offset](#) property value doesn't equal [TimeSpan.Zero](#), it converts the time to UTC.

① Note

If your application requires that converted [DateTime](#) values unambiguously identify a single point in time, you should consider using the [DateTimeOffset.UtcDateTime](#) property to handle all [DateTimeOffset](#) to [DateTime](#) conversions.

The following code uses the [UtcDateTime](#) property to convert a [DateTimeOffset](#) value whose offset equals [TimeSpan.Zero](#) to a [DateTime](#) value:

C#

```
DateTimeOffset utcTime1 = new DateTimeOffset(2008, 6, 19, 7, 0, 0,
TimeSpan.Zero);
DateTime utcTime2 = utcTime1.UtcDateTime;
Console.WriteLine("{0} converted to {1} {2}",
    utcTime1,
    utcTime2,
    utcTime2.Kind);
// The example displays the following output to the console:
//   6/19/2008 7:00:00 AM +00:00 converted to 6/19/2008 7:00:00 AM Utc
```

The following code uses the [UtcDateTime](#) property to perform both a time zone conversion and a type conversion on a [DateTimeOffset](#) value:

C#

```
DateTimeOffset originalTime = new DateTimeOffset(2008, 6, 19, 7, 0, 0, new
TimeSpan(5, 0, 0));
DateTime utcTime = originalTime.UtcDateTime;
Console.WriteLine("{0} converted to {1} {2}",
    originalTime,
    utcTime,
    utcTime.Kind);
// The example displays the following output to the console:
//   6/19/2008 7:00:00 AM +05:00 converted to 6/19/2008 2:00:00 AM Utc
```

Converting a local time

To indicate that a [DateTimeOffset](#) value represents the local time, you can pass the [DateTime](#) value returned by the [DateTimeOffset.DateTime](#) property to the `static` (`Shared` in Visual Basic) [SpecifyKind](#) method. The method returns the date and time passed to it as its first parameter but sets the [Kind](#) property to the value specified by its second parameter. The following code uses the [SpecifyKind](#) method when converting a [DateTimeOffset](#) value whose offset corresponds to that of the local time zone:

C#

```
DateTime sourceDate = new DateTime(2008, 6, 19, 7, 0, 0);
DateTimeOffset utcTime1 = new DateTimeOffset(sourceDate,
                                             TimeZoneInfo.Local.GetUtcOffset(sourceDate));
DateTime utcTime2 = utcTime1.DateTime;
if
(utcTime1.Offset.Equals(TimeZoneInfo.Local.GetUtcOffset(utcTime1.DateTime)))
    utcTime2 = DateTime.SpecifyKind(utcTime2, DateTimeKind.Local);

Console.WriteLine("{0} converted to {1} {2}",
                  utcTime1,
                  utcTime2,
                  utcTime2.Kind);
// The example displays the following output to the console:
//   6/19/2008 7:00:00 AM -07:00 converted to 6/19/2008 7:00:00 AM Local
```

You can also use the [DateTimeOffset.LocalDateTime](#) property to convert a [DateTimeOffset](#) value to a local [DateTime](#) value. The [Kind](#) property of the returned [DateTime](#) value is [Local](#). The following code uses the [DateTimeOffset.LocalDateTime](#) property when converting a [DateTimeOffset](#) value whose offset corresponds to that of the local time zone:

C#

```
DateTime sourceDate = new DateTime(2008, 6, 19, 7, 0, 0);
DateTimeOffset localTime1 = new DateTimeOffset(sourceDate,
                                               TimeZoneInfo.Local.GetUtcOffset(sourceDate));
DateTime localTime2 = localTime1.LocalDateTime;

Console.WriteLine("{0} converted to {1} {2}",
                  localTime1,
                  localTime2,
                  localTime2.Kind);
// The example displays the following output to the console:
//   6/19/2008 7:00:00 AM -07:00 converted to 6/19/2008 7:00:00 AM Local
```

When you retrieve a [DateTime](#) value using the [DateTimeOffset.LocalDateTime](#) property, the property's `get` accessor first converts the [DateTimeOffset](#) value to UTC, then converts it to local time by calling the [ToLocalTime](#) method. This behavior means that you can retrieve a value from the [DateTimeOffset.LocalDateTime](#) property to perform a time zone conversion at the same time that you perform a type conversion. It also means that the local time zone's adjustment rules are applied in performing the conversion. The following code illustrates the use of the [DateTimeOffset.LocalDateTime](#) property to perform both a type and a time zone conversion. The sample output is for a machine set to the Pacific Time Zone (US and Canada). The November date is Pacific Standard Time, which is UTC-8, while the June date is Daylight Savings Time, which is UTC-7.

C#

```
DateTimeOffset originalDate;
DateTime localDate;

// Convert time originating in a different time zone
originalDate = new DateTimeOffset(2008, 6, 18, 7, 0, 0,
                                  new TimeSpan(-5, 0, 0));
localDate = originalDate.LocalDateTime;
Console.WriteLine("{0} converted to {1} {2}",
                  originalDate,
                  localDate,
                  localDate.Kind);

// Convert time originating in a different time zone
// so local time zone's adjustment rules are applied
originalDate = new DateTimeOffset(2007, 11, 4, 4, 0, 0,
                                  new TimeSpan(-5, 0, 0));
localDate = originalDate.LocalDateTime;
Console.WriteLine("{0} converted to {1} {2}",
                  originalDate,
                  localDate,
                  localDate.Kind);

// The example displays the following output to the console,
// when you run it on a machine that is set to Pacific Time (US & Canada):
//       6/18/2008 7:00:00 AM -05:00 converted to 6/18/2008 5:00:00 AM Local
//       11/4/2007 4:00:00 AM -05:00 converted to 11/4/2007 1:00:00 AM Local
```

A general-purpose conversion method

The following example defines a method named `ConvertFromDateTimeOffset` that converts [DateTimeOffset](#) values to [DateTime](#) values. Based on its offset, it determines whether the [DateTimeOffset](#) value is a UTC time, a local time, or some other time and defines the returned date and time value's [Kind](#) property accordingly.

C#

```
static DateTime ConvertFromDateTimeOffset(DateTimeOffset dateTime)
{
    if (dateTime.Offset.Equals(TimeSpan.Zero))
        return dateTime.UtcDateTime;
    else if
(dateTime.Offset.Equals(TimeZoneInfo.Local.GetUtcOffset(dateTime.DateTime)))
        return DateTime.SpecifyKind(dateTime.DateTime, DateTimeKind.Local);
    else
        return dateTime.DateTime;
}
```

The following example calls the `ConvertFromDateTimeOffset` method to convert `DateTimeOffset` values that represent a UTC time, a local time, and a time in the U.S. Central Standard Time zone.

C#

```
DateTime timeComponent = new DateTime(2008, 6, 19, 7, 0, 0);
DateTime returnedDate;

// Convert UTC time
DateTimeOffset utcTime = new DateTimeOffset(timeComponent, TimeSpan.Zero);
returnedDate = ConvertFromDateTimeOffset(utcTime);
Console.WriteLine("{0} converted to {1} {2}",
    utcTime,
    returnedDate,
    returnedDate.Kind);

// Convert local time
DateTimeOffset localTime = new DateTimeOffset(timeComponent,
    TimeZoneInfo.Local.GetUtcOffset(timeComponent));
returnedDate = ConvertFromDateTimeOffset(localTime);
Console.WriteLine("{0} converted to {1} {2}",
    localTime,
    returnedDate,
    returnedDate.Kind);

// Convert Central Standard Time
DateTimeOffset cstTime = new DateTimeOffset(timeComponent,
    TimeZoneInfo.FindSystemTimeZoneById("Central Standard
Time").GetUtcOffset(timeComponent));
returnedDate = ConvertFromDateTimeOffset(cstTime);
Console.WriteLine("{0} converted to {1} {2}",
    cstTime,
    returnedDate,
    returnedDate.Kind);

// The example displays the following output to the console:
//   6/19/2008 7:00:00 AM +00:00 converted to 6/19/2008 7:00:00 AM Utc
//   6/19/2008 7:00:00 AM -07:00 converted to 6/19/2008 7:00:00 AM Local
```

```
// 6/19/2008 7:00:00 AM -05:00 converted to 6/19/2008 7:00:00 AM  
Unspecified
```

ⓘ Note

The code makes the following two assumptions, depending on the application and the source of its date and time values, might not always be valid:

- It assumes that a date and time value whose offset is `TimeSpan.Zero` represents UTC. In fact, UTC isn't a time in a particular time zone, but the time in relation to which the times in the world's time zones are standardized. Time zones can also have an offset of `Zero`.
- It assumes that a date and time whose offset equals that of the local time zone represents the local time zone. Because date and time values are disassociated from their original time zone, this might not be the case; the date and time can have originated in another time zone with the same offset.

See also

- [Dates, times, and time zones](#)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback [here](#).

 [Open a documentation issue](#)

 [Provide product feedback](#)

Converting times between time zones

Article • 09/08/2022

It's becoming increasingly important for any application that works with dates and times to handle differences between time zones. An application can no longer assume that all times can be expressed in the local time, which is the time available from the [DateTime](#) structure. For example, a web page that displays the current time in the eastern part of the United States will lack credibility to a customer in eastern Asia. This article explains how to convert times from one time zone to another and convert [DateTimeOffset](#) values that have limited time zone awareness.

Converting to Coordinated Universal Time

Coordinated Universal Time (UTC) is a high-precision, atomic time standard. The world's time zones are expressed as positive or negative offsets from UTC. Thus, UTC provides a time-zone free or time-zone neutral time. The use of UTC is recommended when a date and time's portability across computers is important. For details and other best practices using dates and times, see [Coding best practices using DateTime in the .NET Framework](#). Converting individual time zones to UTC makes time comparisons easy.

ⓘ Note

You can also serialize a [DateTimeOffset](#) structure to represent a single point in time unambiguously. Because [DateTimeOffset](#) objects store a date and time value along with its offset from UTC, they always represent a particular point in time in relation to UTC.

The easiest way to convert a time to UTC is to call the `static (Shared in Visual Basic)` [TimeZoneInfo.ConvertTimeToUtc\(DateTime\)](#) method. The exact conversion performed by the method depends on the value of the `dateTime` parameter's [Kind](#) property, as the following table shows:

| <code>DateTime.Kind</code> | Conversion |
|---------------------------------------|---|
| <code>DateTimeKind.Local</code> | Converts local time to UTC. |
| <code>DateTimeKind.Unspecified</code> | Assumes the <code>dateTime</code> parameter is local time and converts local time to UTC. |
| <code>DateTimeKind.Utc</code> | Returns the <code>dateTime</code> parameter unchanged. |

The following code converts the current local time to UTC and displays the result to the console:

```
C#
```

```
DateTime dateNow = DateTime.Now;
Console.WriteLine("The date and time are {0} UTC.",
    TimeZoneInfo.ConvertTimeToUtc(dateNow));
```

If the date and time value doesn't represent the local time or UTC, the [ToUniversalTime](#) method will likely return an erroneous result. However, you can use the [TimeZoneInfo.ConvertTimeToUtc](#) method to convert the date and time from a specified time zone. For details on retrieving a [TimeZoneInfo](#) object that represents the destination time zone, see [Finding the time zones defined on a local system](#). The following code uses the [TimeZoneInfo.ConvertTimeToUtc](#) method to convert Eastern Standard Time to UTC:

```
C#
```

```
DateTime easternTime = new DateTime(2007, 01, 02, 12, 16, 00);
string easternZoneId = "Eastern Standard Time";
try
{
    TimeZoneInfo easternZone =
    TimeZoneInfo.FindSystemTimeZoneById(easternZoneId);
    Console.WriteLine("The date and time are {0} UTC.",
        TimeZoneInfo.ConvertTimeToUtc(easternTime,
easternZone));
}
catch (TimeZoneNotFoundException)
{
    Console.WriteLine("Unable to find the {0} zone in the registry.",
        easternZoneId);
}
catch (InvalidTimeZoneException)
{
    Console.WriteLine("Registry data on the {0} zone has been corrupted.",
        easternZoneId);
}
```

The [TimeZoneInfo.ConvertTimeToUtc](#) method throws an [ArgumentException](#) if the [DateTime](#) object's [Kind](#) property and the time zone are mismatched. A mismatch occurs if the [Kind](#) property is [DateTimeKind.Local](#) but the [TimeZoneInfo](#) object doesn't represent the local time zone, or if the [Kind](#) property is [DateTimeKind.Utc](#) but the [TimeZoneInfo](#) object doesn't equal [TimeZoneInfo.Utc](#).

All of these methods take [DateTime](#) values as parameters and return a [DateTime](#) value. For [DateTimeOffset](#) values, the [DateTimeOffset](#) structure has a [ToUniversalTime](#) instance method that converts the date and time of the current instance to UTC. The following example calls the [ToUniversalTime](#) method to convert a local time and several other times to UTC:

C#

```
DateTimeOffset localTime, otherTime, universalTime;

// Define local time in local time zone
localTime = new DateTimeOffset(new DateTime(2007, 6, 15, 12, 0, 0));
Console.WriteLine("Local time: {0}", localTime);
Console.WriteLine();

// Convert local time to offset 0 and assign to otherTime
otherTime = localTime.ToOffset(TimeSpan.Zero);
Console.WriteLine("Other time: {0}", otherTime);
Console.WriteLine("{0} = {1}: {2}",
    localTime, otherTime,
    localTime.Equals(otherTime));
Console.WriteLine("{0} exactly equals {1}: {2}",
    localTime, otherTime,
    localTime.EqualsExact(otherTime));
Console.WriteLine();

// Convert other time to UTC
universalTime = localTime.ToUniversalTime();
Console.WriteLine("Universal time: {0}", universalTime);
Console.WriteLine("{0} = {1}: {2}",
    otherTime, universalTime,
    universalTime.Equals(otherTime));
Console.WriteLine("{0} exactly equals {1}: {2}",
    otherTime, universalTime,
    universalTime.EqualsExact(otherTime));
Console.WriteLine();

// The example produces the following output to the console:
// Local time: 6/15/2007 12:00:00 PM -07:00
//
// Other time: 6/15/2007 7:00:00 PM +00:00
// 6/15/2007 12:00:00 PM -07:00 = 6/15/2007 7:00:00 PM +00:00: True
// 6/15/2007 12:00:00 PM -07:00 exactly equals 6/15/2007 7:00:00 PM
// +00:00: False
//
// Universal time: 6/15/2007 7:00:00 PM +00:00
// 6/15/2007 7:00:00 PM +00:00 = 6/15/2007 7:00:00 PM +00:00: True
// 6/15/2007 7:00:00 PM +00:00 exactly equals 6/15/2007 7:00:00 PM
// +00:00: True
```

Converting UTC to a designated time zone

To convert UTC to local time, see the [Converting UTC to local time](#) section that follows.

To convert UTC to the time in any time zone that you designate, call the

[ConvertTimeFromUtc](#) method. The method takes two parameters:

- The UTC to convert. This must be a [DateTime](#) value whose [Kind](#) property is set to [Unspecified](#) or [Utc](#).
- The time zone to convert the UTC to.

The following code converts UTC to Central Standard Time:

C#

```
DateTime timeUtc = DateTime.UtcNow;
try
{
    TimeZoneInfo cstZone = TimeZoneInfo.FindSystemTimeZoneById("Central
Standard Time");
    DateTime cstTime = TimeZoneInfo.ConvertTimeFromUtc(timeUtc, cstZone);
    Console.WriteLine("The date and time are {0} {1}.",
                      cstTime,
                      cstZone.IsDaylightSavingTime(cstTime) ?
                        cstZone.DaylightName : cstZone.StandardName);
}
catch (TimeZoneNotFoundException)
{
    Console.WriteLine("The registry does not define the Central Standard Time
zone.");
}
catch (InvalidTimeZoneException)
{
    Console.WriteLine("Registry data on the Central Standard Time zone has
been corrupted.");
}
```

Converting UTC to local time

To convert UTC to local time, call the [ToLocalTime](#) method of the [DateTime](#) object whose time you want to convert. The exact behavior of the method depends on the value of the object's [Kind](#) property, as the following table shows:

| DateTime.Kind | Conversion |
|--|--|
| DateTimeKind.Local | Returns the DateTime value unchanged. |
| DateTimeKind.Unspecified | Assumes that the DateTime value is UTC and converts the UTC to local time. |

| <code>DateTime.Kind</code> | Conversion |
|-------------------------------|--|
| <code>DateTimeKind.Utc</code> | Converts the DateTime value to local time. |

① Note

The [TimeZone.ToLocalTime](#) method behaves identically to the `DateTime.ToLocalTime` method. It takes a single parameter, which is the date and time value, to convert.

You can also convert the time in any designated time zone to local time by using the `static` (`Shared` in Visual Basic) [TimeZoneInfo.ConvertTime](#) method. This technique is discussed in the next section.

Converting between any two time zones

You can convert between any two time zones by using either of the following two `static` (`Shared` in Visual Basic) methods of the [TimeZoneInfo](#) class:

- [ConvertTime](#)

This method's parameters are the date and time value to convert, a `TimeZoneInfo` object that represents the time zone of the date and time value, and a `TimeZoneInfo` object that represents the time zone to convert the date and time value to.

- [ConvertTimeBySystemTimeZoneld](#)

This method's parameters are the date and time value to convert, the identifier of the date and time value's time zone, and the identifier of the time zone to convert the date and time value to.

Both methods require that the `Kind` property of the date and time value to convert and the [TimeZoneInfo](#) object or time zone identifier that represents its time zone correspond to one another. Otherwise, an [ArgumentException](#) is thrown. For example, if the `Kind` property of the date and time value is `DateTimeKind.Local`, an exception is thrown if the `TimeZoneInfo` object passed as a parameter to the method isn't equal to `TimeZoneInfo.Local`. An exception is also thrown if the identifier passed as a parameter to the method isn't equal to `TimeZoneInfo.Local.Id`.

The following example uses the [ConvertTime](#) method to convert from Hawaiian Standard Time to local time:

```
C#  
  
DateTime hwTime = new DateTime(2007, 02, 01, 08, 00, 00);  
try  
{  
    TimeZoneInfo hwZone = TimeZoneInfo.FindSystemTimeZoneById("Hawaiian  
Standard Time");  
    Console.WriteLine("{0} {1} is {2} local time.",  
        hwTime,  
        hwZone.IsDaylightSavingTime(hwTime) ? hwZone.DaylightName :  
        hwZone.StandardName,  
        TimeZoneInfo.ConvertTime(hwTime, hwZone, TimeZoneInfo.Local));  
}  
catch (TimeZoneNotFoundException)  
{  
    Console.WriteLine("The registry does not define the Hawaiian Standard  
Time zone.");  
}  
catch (InvalidTimeZoneException)  
{  
    Console.WriteLine("Registry data on the Hawaiian Standard Time zone has  
been corrupted.");  
}
```

Converting `DateTimeOffset` values

Date and time values represented by [DateTimeOffset](#) objects aren't fully time-zone aware because the object is disassociated from its time zone at the time it's instantiated. However, in many cases, an application simply needs to convert a date and time based on two different offsets from UTC rather than on time in particular time zones. To perform this conversion, you can call the current instance's [ToOffset](#) method. The method's single parameter is the offset of the new date and time value the method will return.

For example, if the date and time of a user request for a web page is known and is serialized as a string in the format MM/dd/yyyy hh:mm:ss zzzz, the following `ReturnTimeOnServer` method converts this date and time value to the date and time on the web server:

```
C#  
  
public DateTimeOffset ReturnTimeOnServer(string clientString)  
{  
    string format = @"M/d/yyyy H:m:s zzz";
```

```
    TimeSpan serverOffset =
TimeZoneInfo.Local.GetUtcOffset(DateTimeOffset.Now);

    try
    {
        DateTimeOffset clientTime = DateTimeOffset.ParseExact(clientString,
format, CultureInfo.InvariantCulture);
        DateTimeOffset serverTime = clientTime.ToOffset(serverOffset);
        return serverTime;
    }
    catch (FormatException)
    {
        return DateTimeOffset.MinValue;
    }
}
```

If the method passes the string "9/1/2007 5:32:07 -05:00," which represents the date and time in a time zone five hours earlier than UTC, it returns "9/1/2007 3:32:07 AM -07:00" for a server located in the U.S. Pacific Standard Time zone.

The [TimeZoneInfo](#) class also includes an overload of the [TimeZoneInfo.ConvertTime\(DateTimeOffset, TimeZoneInfo\)](#) method that performs time zone conversions with [ToOffset\(TimeSpan\)](#) values. The method's parameters are a [DateTimeOffset](#) value and a reference to the time zone to which the time is to be converted. The method call returns a [DateTimeOffset](#) value. For example, the [ReturnTimeOnServer](#) method in the previous example could be rewritten as follows to call the [ConvertTime\(DateTimeOffset, TimeZoneInfo\)](#) method.

C#

```
public DateTimeOffset ReturnTimeOnServer(string clientString)
{
    string format = @"M/d/yyyy H:m:s zzz";

    try
    {
        DateTimeOffset clientTime = DateTimeOffset.ParseExact(clientString,
format,
                           CultureInfo.InvariantCulture);
        DateTimeOffset serverTime = TimeZoneInfo.ConvertTime(clientTime,
TimeZoneInfo.Local);
        return serverTime;
    }
    catch (FormatException)
    {
        return DateTimeOffset.MinValue;
    }
}
```

See also

- [TimeZoneInfo](#)
- [Dates, times, and time zones](#)
- [Finding the time zones defined on a local system](#)

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

How to: Resolve ambiguous times

Article • 09/15/2021

An ambiguous time is a time that maps to more than one Coordinated Universal Time (UTC). It occurs when the clock time is adjusted back in time, such as during the transition from a time zone's daylight saving time to its standard time. When handling an ambiguous time, you can do one of the following:

- Make an assumption about how the time maps to UTC. For example, you can assume that an ambiguous time is always expressed in the time zone's standard time.
- If the ambiguous time is an item of data entered by the user, you can leave it to the user to resolve the ambiguity.

This topic shows how to resolve an ambiguous time by assuming that it represents the time zone's standard time.

To map an ambiguous time to a time zone's standard time

1. Call the [IsAmbiguousTime](#) method to determine whether the time is ambiguous.
2. If the time is ambiguous, subtract the time from the [TimeSpan](#) object returned by the time zone's [BaseUtcOffset](#) property.
3. Call the `static` (`Shared` in Visual Basic .NET) [SpecifyKind](#) method to set the UTC date and time value's [Kind](#) property to [DateTimeKind.Utc](#).

Example

The following example illustrates how to convert an ambiguous time to UTC by assuming that it represents the local time zone's standard time.

C#

```
private DateTime ResolveAmbiguousTime(DateTime ambiguousTime)
{
    // Time is not ambiguous
    if (!TimeZoneInfo.Local.IsAmbiguousTime(ambiguousTime))
    {
        return ambiguousTime;
    }
```

```

// Time is ambiguous
else
{
    DateTime utcTime = DateTime.SpecifyKind(ambiguousTime -
TimeZoneInfo.Local.BaseUtcOffset,
                                         DateTimeKind.Utc);
    Console.WriteLine("{0} local time corresponds to {1} {2}.",
                      ambiguousTime, utcTime, utcTime.Kind.ToString());
    return utcTime;
}
}

```

The example consists of a method named `ResolveAmbiguousTime` that determines whether the `DateTime` value passed to it is ambiguous. If the value is ambiguous, the method returns a `DateTime` value that represents the corresponding UTC time. The method handles this conversion by subtracting the value of the local time zone's `BaseUtcOffset` property from the local time.

Ordinarily, an ambiguous time is handled by calling the `GetAmbiguousTimeOffsets` method to retrieve an array of `TimeSpan` objects that contain the ambiguous time's possible UTC offsets. However, this example makes the arbitrary assumption that an ambiguous time should always be mapped to the time zone's standard time. The `BaseUtcOffset` property returns the offset between UTC and a time zone's standard time.

In this example, all references to the local time zone are made through the `TimeZoneInfo.Local` property; the local time zone is never assigned to an object variable. This is a recommended practice because a call to the `TimeZoneInfo.ClearCachedData` method invalidates any objects that the local time zone is assigned to.

Compiling the code

This example requires:

- That the `System` namespace be imported with the `using` statement (required in C# code).

See also

- [Dates, times, and time zones](#)
- [How to: Let users resolve ambiguous times](#)

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

How to: Let users resolve ambiguous times

Article • 09/15/2021

An ambiguous time is a time that maps to more than one Coordinated Universal Time (UTC). It occurs when the clock time is adjusted back in time, such as during the transition from a time zone's daylight saving time to its standard time. When handling an ambiguous time, you can do one of the following:

- If the ambiguous time is an item of data entered by the user, you can leave it to the user to resolve the ambiguity.
- Make an assumption about how the time maps to UTC. For example, you can assume that an ambiguous time is always expressed in the time zone's standard time.

This topic shows how to let a user resolve an ambiguous time.

To let a user resolve an ambiguous time

1. Get the date and time input by the user.
2. Call the [IsAmbiguousTime](#) method to determine whether the time is ambiguous.
3. If the time is ambiguous, call the [GetAmbiguousTimeOffsets](#) method to retrieve an array of [TimeSpan](#) objects. Each element in the array contains a UTC offset that the ambiguous time can map to.
4. Let the user select the desired offset.
5. Get the UTC date and time by subtracting the offset selected by the user from the local time.
6. Call the `static` (`Shared` in Visual Basic .NET) [SpecifyKind](#) method to set the UTC date and time value's [Kind](#) property to [DateTimeKind.Utc](#).

Example

The following example prompts the user to enter a date and time and, if it is ambiguous, lets the user select the UTC time that the ambiguous time maps to.

C#

```

private void GetUserDateInput()
{
    // Get date and time from user
    DateTime inputDate = GetUserDateTime();
    DateTime utcDate;

    // Exit if date has no significant value
    if (inputDate == DateTime.MinValue) return;

    if (TimeZoneInfo.Local.IsAmbiguousTime(inputDate))
    {
        Console.WriteLine("The date you've entered is ambiguous.");
        Console.WriteLine("Please select the correct offset from Universal
Coordinated Time:");
        TimeSpan[] offsets =
            TimeZoneInfo.Local.GetAmbiguousTimeOffsets(inputDate);
        for (int ctr = 0; ctr < offsets.Length; ctr++)
        {
            Console.WriteLine("{0}.) {1} hours, {2} minutes", ctr,
offsets[ctr].Hours, offsets[ctr].Minutes);
        }
        Console.Write("> ");
        int selection = Convert.ToInt32(Console.ReadLine());

        // Convert local time to UTC, and set Kind property to
        DateTimeKind.Utc
        utcDate = DateTime.SpecifyKind(inputDate - offsets[selection],
DateTimeKind.Utc);

        Console.WriteLine("{0} local time corresponds to {1} {2}.",
inputDate,
utcDate, utcDate.Kind.ToString());
    }
    else
    {
        utcDate = inputDate.ToUniversalTime();
        Console.WriteLine("{0} local time corresponds to {1} {2}.",
inputDate,
utcDate, utcDate.Kind.ToString());
    }
}

private DateTime GetUserDateTime()
{
    bool exitFlag = false;           // flag to exit loop if date is valid
    string dateString;
    DateTime inputDate = DateTime.MinValue;

    Console.Write("Enter a local date and time: ");
    while (!exitFlag)
    {
        dateString = Console.ReadLine();
        if (dateString.ToUpper() == "E")
            exitFlag = true;

        if (DateTime.TryParse(dateString, out inputDate))

```

```

        exitFlag = true;
    else
        Console.WriteLine("Enter a valid date and time, or enter 'e' to exit:");
    ");
}

return inputDate;
}

```

VB

```

Private Sub GetUserDateInput()
    ' Get date and time from user
    Dim inputDate As Date = GetUserDateTime()
    Dim utcDate As Date

    ' Exit if date has no significant value
    If inputDate = Date.MinValue Then Exit Sub

    If TimeZoneInfo.Local.IsAmbiguousTime(inputDate) Then
        Console.WriteLine("The date you've entered is ambiguous.")
        Console.WriteLine("Please select the correct offset from Universal
Coordinated Time:")
        Dim offsets() As TimeSpan =
TimeZoneInfo.Local.GetAmbiguousTimeOffsets(inputDate)
        For ctr As Integer = 0 to offsets.Length - 1
            Dim zoneDescription As String
            If offsets(ctr).Equals(TimeZoneInfo.Local.BaseUtcOffset) Then
                zoneDescription = TimeZoneInfo.Local.StandardName
            Else
                zoneDescription = TimeZoneInfo.Local.DaylightName
            End If
            Console.WriteLine("{0}.) {1} hours, {2} minutes ({3})",
                ctr, offsets(ctr).Hours, offsets(ctr).Minutes,
                zoneDescription)
        Next
        Console.Write("> ")
        Dim selection As Integer = CInt(Console.ReadLine())

        ' Convert local time to UTC, and set Kind property to
        DateTimeKind.Utc
        utcDate = Date.SpecifyKind(inputDate - offsets(selection),
        DateTimeKind.Utc)

        Console.WriteLine("{0} local time corresponds to {1} {2}.",
            inputDate, utcDate, utcDate.Kind.ToString())
    Else
        utcDate = inputDate.ToUniversalTime()
        Console.WriteLine("{0} local time corresponds to {1} {2}.",
            inputDate, utcDate, utcDate.Kind.ToString())
    End If
End Sub

```

```

Private Function GetUserDateTime() As Date
    Dim exitFlag As Boolean = False           ' flag to exit loop if date
is valid
    Dim dateString As String
    Dim inputDate As Date = Date.MinValue

    Console.Write("Enter a local date and time: ")
    Do While Not exitFlag
        dateString = Console.ReadLine()
        If dateString.ToUpper = "E" Then exitFlag = True
        If Date.TryParse(dateString, inputDate) Then
            exitFlag = true
        Else
            Console.Write("Enter a valid date and time, or enter 'e' to
exit: ")
        End If
    Loop

    Return inputDate
End Function

```

The core of the example code uses an array of [TimeSpan](#) objects to indicate possible offsets of the ambiguous time from UTC. However, these offsets are unlikely to be meaningful to the user. To clarify the meaning of the offsets, the code also notes whether an offset represents the local time zone's standard time or its daylight saving time. The code determines which time is standard and which time is daylight by comparing the offset with the value of the [BaseUtcOffset](#) property. This property indicates the difference between the UTC and the time zone's standard time.

In this example, all references to the local time zone are made through the [TimeZoneInfo.Local](#) property; the local time zone is never assigned to an object variable. This is a recommended practice because a call to the [TimeZoneInfo.ClearCachedData](#) method invalidates any objects that the local time zone is assigned to.

Compiling the code

This example requires:

- That the [System](#) namespace be imported with the `using` statement (required in C# code).

See also

- [Dates, times, and time zones](#)
- [How to: Resolve ambiguous times](#)

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

Instantiating a DateTimeOffset object

Article • 01/03/2023

The [DateTimeOffset](#) structure offers a number of ways to create new [DateTimeOffset](#) values. Many of them correspond directly to the methods available for instantiating new [DateTime](#) values, with enhancements that allow you to specify the date and time value's offset from Coordinated Universal Time (UTC). In particular, you can instantiate a [DateTimeOffset](#) value in the following ways:

- By using a date and time literal.
- By calling a [DateTimeOffset](#) constructor.
- By implicitly converting a value to [DateTimeOffset](#) value.
- By parsing the string representation of a date and time.

This topic provides greater detail and code examples that illustrate these methods of instantiating new [DateTimeOffset](#) values.

Date and time literals

For languages that support it, one of the most common ways to instantiate a [DateTime](#) value is to provide the date and time as a hard-coded literal value. For example, the following Visual Basic code creates a [DateTime](#) object whose value is May 1, 2008, at 8:06:32 AM.

VB

```
Dim literalDate1 As Date = #05/01/2008 8:06:32 AM#
Console.WriteLine(literalDate1.ToString())
' Displays:
'      5/1/2008 8:06:32 AM
```

[DateTimeOffset](#) values can also be initialized using date and time literals when using languages that support [DateTime](#) literals. For example, the following Visual Basic code creates a [DateTimeOffset](#) object.

VB

```
Dim literalDate As DateTimeOffset = #05/01/2008 8:06:32 AM#
Console.WriteLine(literalDate.ToString())
```

```
' Displays:  
' 5/1/2008 8:06:32 AM -07:00
```

As the console output shows, the [DateTimeOffset](#) value created in this way is assigned the offset of the local time zone. This means that a [DateTimeOffset](#) value assigned using a character literal does not identify a single point of time if the code is run on different computers.

DateTimeOffset constructors

The [DateTimeOffset](#) type defines six constructors. Four of them correspond directly to [DateTime](#) constructors, with an additional parameter of type [TimeSpan](#) that defines the date and time's offset from UTC. These allow you to define a [DateTimeOffset](#) value based on the value of its individual date and time components. For example, the following code uses these four constructors to instantiate [DateTimeOffset](#) objects with identical values of 5/1/2008 8:06:32 +01:00.

C#

```
DateTimeOffset dateAndTime;  
  
// Instantiate date and time using years, months, days,  
// hours, minutes, and seconds  
dateAndTime = new DateTimeOffset(2008, 5, 1, 8, 6, 32,  
                               new TimeSpan(1, 0, 0));  
Console.WriteLine(dateAndTime);  
// Instantiate date and time using years, months, days,  
// hours, minutes, seconds, and milliseconds  
dateAndTime = new DateTimeOffset(2008, 5, 1, 8, 6, 32, 545,  
                               new TimeSpan(1, 0, 0));  
Console.WriteLine("{0} {1}", dateAndTime.ToString("G"),  
                  dateAndTime.ToString("zzz"));  
  
// Instantiate date and time using Persian calendar with years,  
// months, days, hours, minutes, seconds, and milliseconds  
dateAndTime = new DateTimeOffset(1387, 2, 12, 8, 6, 32, 545,  
                               new PersianCalendar(),  
                               new TimeSpan(1, 0, 0));  
// Note that the console output displays the date in the Gregorian  
// calendar, not the Persian calendar.  
Console.WriteLine("{0} {1}", dateAndTime.ToString("G"),  
                  dateAndTime.ToString("zzz"));  
  
// Instantiate date and time using number of ticks  
// 05/01/2008 8:06:32 AM is 633,452,259,920,000,000 ticks  
dateAndTime = new DateTimeOffset(633452259920000000, new TimeSpan(1, 0, 0));  
Console.WriteLine(dateAndTime);  
// The example displays the following output to the console:  
//      5/1/2008 8:06:32 AM +01:00
```

```
//      5/1/2008 8:06:32 AM +01:00
//      5/1/2008 8:06:32 AM +01:00
//      5/1/2008 8:06:32 AM +01:00
```

Note that, when the value of the [DateTimeOffset](#) object instantiated using a [PersianCalendar](#) object as one of the arguments to its constructor is displayed to the console, it is expressed as a date in the Gregorian rather than the Persian calendar. To output a date using the Persian calendar, see the example in the [PersianCalendar](#) topic.

The other two constructors create a [DateTimeOffset](#) object from a [DateTime](#) value. The first of these has a single parameter, the [DateTime](#) value to convert to a [DateTimeOffset](#) value. The offset of the resulting [DateTimeOffset](#) value depends on the [Kind](#) property of the constructor's single parameter. If its value is [DateTimeKind.Utc](#), the offset is set equal to [TimeSpan.Zero](#). Otherwise, its offset is set equal to that of the local time zone. The following example illustrates the use of this constructor to instantiate [DateTimeOffset](#) objects representing UTC and the local time zone:

C#

```
// Declare date; Kind property is DateTimeKind.Unspecified
DateTime sourceDate = new DateTime(2008, 5, 1, 8, 30, 0);
DateTimeOffset targetTime;

// Instantiate a DateTimeOffset value from a UTC time
DateTime utcTime = DateTime.SpecifyKind(sourceDate, DateTimeKind.Utc);
targetTime = new DateTimeOffset(utcTime);
Console.WriteLine(targetTime);
// Displays 5/1/2008 8:30:00 AM +00:00
// Because the Kind property is DateTimeKind.Utc,
// the offset is TimeSpan.Zero.

// Instantiate a DateTimeOffset value from a UTC time with a zero offset
targetTime = new DateTimeOffset(utcTime, TimeSpan.Zero);
Console.WriteLine(targetTime);
// Displays 5/1/2008 8:30:00 AM +00:00
// Because the Kind property is DateTimeKind.Utc,
// the call to the constructor succeeds

// Instantiate a DateTimeOffset value from a UTC time with a negative offset
try
{
    targetTime = new DateTimeOffset(utcTime, new TimeSpan(-2, 0, 0));
    Console.WriteLine(targetTime);
}
catch (ArgumentException)
{
    Console.WriteLine("Attempt to create DateTimeOffset value from {0} failed.",
                    targetTime);
}
```

```
// Throws exception and displays the following to the console:  
//   Attempt to create DateTimeOffset value from 5/1/2008 8:30:00 AM +00:00  
// failed.  
  
// Instantiate a DateTimeOffset value from a local time  
DateTime localTime = DateTime.SpecifyKind(sourceDate, DateTimeKind.Local);  
targetTime = new DateTimeOffset(localTime);  
Console.WriteLine(targetTime);  
// Displays 5/1/2008 8:30:00 AM -07:00  
// Because the Kind property is DateTimeKind.Local,  
// the offset is that of the local time zone.  
  
// Instantiate a DateTimeOffset value from an unspecified time  
targetTime = new DateTimeOffset(sourceDate);  
Console.WriteLine(targetTime);  
// Displays 5/1/2008 8:30:00 AM -07:00  
// Because the Kind property is DateTimeKind.Unspecified,  
// the offset is that of the local time zone.
```

ⓘ Note

Calling the overload of the `DateTimeOffset` constructor that has a single `DateTime` parameter is equivalent to performing an implicit conversion of a `DateTime` value to a `DateTimeOffset` value.

The second constructor that creates a `DateTimeOffset` object from a `DateTime` value has two parameters: the `DateTime` value to convert, and a `TimeSpan` value representing the date and time's offset from UTC. This offset value must correspond to the `Kind` property of the constructor's first parameter or an `ArgumentException` is thrown. If the `Kind` property of the first parameter is `DateTimeKind.Utc`, the value of the second parameter must be `TimeSpan.Zero`. If the `Kind` property of the first parameter is `DateTimeKind.Local`, the value of the second parameter must be the offset of the local system's time zone. If the `Kind` property of the first parameter is `DateTimeKind.Unspecified`, the offset can be any valid value. The following code illustrates calls to this constructor to convert `DateTime` to `DateTimeOffset` values.

C#

```
DateTime sourceDate = new DateTime(2008, 5, 1, 8, 30, 0);  
DateTimeOffset targetTime;  
  
// Instantiate a DateTimeOffset value from a UTC time with a zero offset.  
DateTime utcTime = DateTime.SpecifyKind(sourceDate, DateTimeKind.Utc);  
targetTime = new DateTimeOffset(utcTime, TimeSpan.Zero);  
Console.WriteLine(targetTime);  
// Displays 5/1/2008 8:30:00 AM +00:00  
// Because the Kind property is DateTimeKind.Utc,
```

```

// the call to the constructor succeeds

// Instantiate a DateTimeOffset value from a UTC time with a non-zero
// offset.
try
{
    targetTime = new DateTimeOffset(utcTime, new TimeSpan(-2, 0, 0));
    Console.WriteLine(targetTime);
}
catch (ArgumentException)
{
    Console.WriteLine("Attempt to create DateTimeOffset value from {0} failed.",
                      utcTime);
}
// Throws exception and displays the following to the console:
//   Attempt to create DateTimeOffset value from 5/1/2008 8:30:00 AM failed.

// Instantiate a DateTimeOffset value from a local time with
// the offset of the local time zone
DateTime localTime = DateTime.SpecifyKind(sourceDate, DateTimeKind.Local);
targetTime = new DateTimeOffset(localTime,
                                TimeZoneInfo.Local.GetUtcOffset(localTime));
Console.WriteLine(targetTime);
// Displays 5/1/2008 8:30:00 AM -07:00
// Because the Kind property is DateTimeKind.Local and the offset matches
// that of the local time zone, the call to the constructor succeeds.

// Instantiate a DateTimeOffset value from a local time with a zero offset.
try
{
    targetTime = new DateTimeOffset(localTime, TimeSpan.Zero);
    Console.WriteLine(targetTime);
}
catch (ArgumentException)
{
    Console.WriteLine("Attempt to create DateTimeOffset value from {0} failed.",
                      localTime);
}
// Throws exception and displays the following to the console:
//   Attempt to create DateTimeOffset value from 5/1/2008 8:30:00 AM failed.

// Instantiate a DateTimeOffset value with an arbitrary time zone.
string timeZoneName = "Central Standard Time";
TimeSpan offset = TimeZoneInfo.FindSystemTimeZoneById(timeZoneName).
    GetUtcOffset(sourceDate);
targetTime = new DateTimeOffset(sourceDate, offset);
Console.WriteLine(targetTime);
// Displays 5/1/2008 8:30:00 AM -05:00

```

Implicit type conversion

The `DateTimeOffset` type supports one *implicit* type conversion: from a `DateTime` value to a `DateTimeOffset` value. (An implicit type conversion is a conversion from one type to another that does not require an explicit cast (in C#) or conversion (in Visual Basic) and that does not lose information.) It makes code like the following possible.

C#

```
DateTimeOffset targetTime;

// The Kind property of sourceDate is DateTimeKind.Unspecified
DateTime sourceDate = new DateTime(2008, 5, 1, 8, 30, 0);
targetTime = sourceDate;
Console.WriteLine(targetTime);
// Displays 5/1/2008 8:30:00 AM -07:00

// define a UTC time (Kind property is DateTimeKind.Utc)
DateTime utcTime = DateTime.SpecifyKind(sourceDate, DateTimeKind.Utc);
targetTime = utcTime;
Console.WriteLine(targetTime);
// Displays 5/1/2008 8:30:00 AM +00:00

// Define a local time (Kind property is DateTimeKind.Local)
DateTime localTime = DateTime.SpecifyKind(sourceDate, DateTimeKind.Local);
targetTime = localTime;
Console.WriteLine(targetTime);
// Displays 5/1/2008 8:30:00 AM -07:00
```

The offset of the resulting `DateTimeOffset` value depends on the `DateTime.Kind` property value. If its value is `DateTimeKind.Utc`, the offset is set equal to `TimeSpan.Zero`. If its value is either `DateTimeKind.Local` or `DateTimeKind.Unspecified`, the offset is set equal to that of the local time zone.

Parsing the string representation of a date and time

The `DateTimeOffset` type supports four methods that allow you to convert the string representation of a date and time into a `DateTimeOffset` value:

- `Parse`, which tries to convert the string representation of a date and time to a `DateTimeOffset` value and throws an exception if the conversion fails.
- `TryParse`, which tries to convert the string representation of a date and time to a `DateTimeOffset` value and returns `false` if the conversion fails.
- `ParseExact`, which tries to convert the string representation of a date and time in a specified format to a `DateTimeOffset` value. The method throws an exception if the

conversion fails.

- [TryParseExact](#), which tries to convert the string representation of a date and time in a specified format to a [DateTimeOffset](#) value. The method returns `false` if the conversion fails.

The following example illustrates calls to each of these four string conversion methods to instantiate a [DateTimeOffset](#) value.

C#

```
string timeString;
DateTimeOffset targetTime;

timeString = "05/01/2008 8:30 AM +01:00";
try
{
    targetTime = DateTimeOffset.Parse(timeString);
    Console.WriteLine(targetTime);
}
catch (FormatException)
{
    Console.WriteLine("Unable to parse {0}.", timeString);
}

timeString = "05/01/2008 8:30 AM";
if (DateTimeOffset.TryParse(timeString, out targetTime))
    Console.WriteLine(targetTime);
else
    Console.WriteLine("Unable to parse {0}.", timeString);

timeString = "Thursday, 01 May 2008 08:30";
try
{
    targetTime = DateTimeOffset.ParseExact(timeString, "f",
        CultureInfo.InvariantCulture);
    Console.WriteLine(targetTime);
}
catch (FormatException)
{
    Console.WriteLine("Unable to parse {0}.", timeString);
}

timeString = "Thursday, 01 May 2008 08:30 +02:00";
string formatString;
formatString = CultureInfo.InvariantCulture.DateTimeFormat.LongDatePattern +
    " " +
    CultureInfo.InvariantCulture.DateTimeFormat.ShortTimePattern +
    " zzz";
if (DateTimeOffset.TryParseExact(timeString,
    formatString,
```

```
        CultureInfo.InvariantCulture,
        DateTimeStyles.AllowLeadingWhite,
        out targetTime))

    Console.WriteLine(targetTime);
else
    Console.WriteLine("Unable to parse {0}.", timeString);
// The example displays the following output to the console:
//      5/1/2008 8:30:00 AM +01:00
//      5/1/2008 8:30:00 AM -07:00
//      5/1/2008 8:30:00 AM -07:00
//      5/1/2008 8:30:00 AM +02:00
```

See also

- [Dates, times, and time zones](#)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

How to: Create time zones without adjustment rules

Article • 09/15/2021

The precise time zone information that is required by an application may not be present on a particular system for several reasons:

- The time zone has never been defined in the local system's registry.
- Data about the time zone has been modified or removed from the registry.
- The time zone exists but does not have accurate information about time zone adjustments for a particular historic period.

In these cases, you can call the [CreateCustomTimeZone](#) method to define the time zone required by your application. You can use the overloads of this method to create a time zone with or without adjustment rules. If the time zone supports daylight saving time, you can define adjustments with either fixed or floating adjustment rules. (For definitions of these terms, see the "Time Zone Terminology" section in [Time zone overview](#).)

Important

Custom time zones created by calling the [CreateCustomTimeZone](#) method are not added to the registry. Instead, they can be accessed only through the object reference returned by the [CreateCustomTimeZone](#) method call.

This topic shows how to create a time zone without adjustment rules. To create a time zone that supports daylight saving time adjustment rules, see [How to: Create time zones with adjustment rules](#).

To create a time zone without adjustment rules

1. Define the time zone's display name.

The display name follows a fairly standard format in which the time zone's offset from Coordinated Universal Time (UTC) is enclosed in parentheses and is followed by a string that identifies the time zone, one or more of the cities in the time zone, or one or more of the countries or regions in the time zone.

2. Define the name of the time zone's standard time. Typically, this string is also used as the time zone's identifier.
3. If you want to use a different identifier than the time zone's standard name, define the time zone identifier.
4. Instantiate a [TimeSpan](#) object that defines the time zone's offset from UTC. Time zones with times that are later than UTC have a positive offset. Time zones with times that are earlier than UTC have a negative offset.
5. Call the [TimeZoneInfo.CreateCustomTimeZone\(String, TimeSpan, String, String\)](#) method to instantiate the new time zone.

Example

The following example defines a custom time zone for Mawson, Antarctica, which has no adjustment rules.

C#

```
string displayName = "(GMT+06:00) Antarctica/Mawson Time";
string standardName = "Mawson Time";
TimeSpan offset = new TimeSpan(06, 00, 00);
TimeZoneInfo mawson = TimeZoneInfo.CreateCustomTimeZone(standardName,
offset, displayName, standardName);
Console.WriteLine("The current time is {0} {1}",
TimeZoneInfo.ConvertTime(DateTime.Now, TimeZoneInfo.Local,
mawson),
mawson.StandardName);
```

The string assigned to the [DisplayName](#) property follows a standard format in which the time zone's offset from UTC is followed by a friendly description of the time zone.

Compiling the code

This example requires:

- That the following namespaces be imported:

C#

```
using System.Collections.Generic;
using System.Collections.ObjectModel;
```

See also

- [Dates, times, and time zones](#)
- [Time zone overview](#)
- [How to: Create time zones with adjustment rules](#)

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

How to: Create time zones with adjustment rules

Article • 09/15/2021

The precise time zone information that is required by an application may not be present on a particular system for several reasons:

- The time zone has never been defined in the local system's registry.
- Data about the time zone has been modified or removed from the registry.
- The time zone does not have accurate information about time zone adjustments for a particular historic period.

In these cases, you can call the [CreateCustomTimeZone](#) method to define the time zone required by your application. You can use the overloads of this method to create a time zone with or without adjustment rules. If the time zone supports daylight saving time, you can define adjustments with either fixed or floating adjustment rules. (For definitions of these terms, see the "Time Zone Terminology" section in [Time zone overview](#).)

ⓘ Important

Custom time zones created by calling the [CreateCustomTimeZone](#) method are not added to the registry. Instead, they can be accessed only through the object reference returned by the [CreateCustomTimeZone](#) method call.

This topic shows how to create a time zone with adjustment rules. To create a time zone that does not support daylight saving time adjustment rules, see [How to: Create Time Zones Without Adjustment Rules](#).

To create a time zone with floating adjustment rules

1. For each adjustment (that is, for each transition away from and back to standard time over a particular time interval), do the following:
 - a. Define the starting transition time for the time zone adjustment.

You must call the [TimeZoneInfo.TransitionTime.CreateFloatingDateRule](#) method and pass it a [DateTime](#) value that defines the time of the transition, an integer value that defines the month of the transition, an integer value that defines the

week on which the transition occurs, and a [DayOfWeek](#) value that defines the day of the week on which the transition occurs. This method call instantiates a [TimeZoneInfo.TransitionTime](#) object.

- b. Define the ending transition time for the time zone adjustment. This requires another call to the [TimeZoneInfo.TransitionTime.CreateFloatingDateRule](#) method. This method call instantiates a second [TimeZoneInfo.TransitionTime](#) object.
 - c. Call the [CreateAdjustmentRule](#) method and pass it the effective start and end dates of the adjustment, a [TimeSpan](#) object that defines the amount of time in the transition, and the two [TimeZoneInfo.TransitionTime](#) objects that define when the transitions to and from daylight saving time occur. This method call instantiates a [TimeZoneInfo.AdjustmentRule](#) object.
 - d. Assign the [TimeZoneInfo.AdjustmentRule](#) object to an array of [TimeZoneInfo.AdjustmentRule](#) objects.
2. Define the time zone's display name. The display name follows a fairly standard format in which the time zone's offset from Coordinated Universal Time (UTC) is enclosed in parentheses and is followed by a string that identifies the time zone, one or more of the cities in the time zone, or one or more of the countries or regions in the time zone.
 3. Define the name of the time zone's standard time. Typically, this string is also used as the time zone's identifier.
 4. Define the name of the time zone's daylight time.
 5. If you want to use a different identifier than the time zone's standard name, define the time zone identifier.
 6. Instantiate a [TimeSpan](#) object that defines the time zone's offset from UTC. Time zones with times that are later than UTC have a positive offset. Time zones with times that are earlier than UTC have a negative offset.
 7. Call the [TimeZoneInfo.CreateCustomTimeZone\(String, TimeSpan, String, String, TimeZoneInfo+AdjustmentRule\[\]\)](#) method to instantiate the new time zone.

Example

The following example defines a Central Standard Time zone for the United States that includes adjustment rules for a variety of time intervals from 1918 to the present.

C#

```
TimeZoneInfo cst;
// Declare necessary TimeZoneInfo.AdjustmentRule objects for time zone
TimeSpan delta = new TimeSpan(1, 0, 0);
TimeZoneInfo.AdjustmentRule adjustment;
List<TimeZoneInfo.AdjustmentRule> adjustmentList = new
List<TimeZoneInfo.AdjustmentRule>();
// Declare transition time variables to hold transition time information
TimeZoneInfo.TransitionTime transitionRuleStart, transitionRuleEnd;

// Define new Central Standard Time zone 6 hours earlier than UTC
// Define rule 1 (for 1918-1919)
transitionRuleStart = TimeZoneInfo.TransitionTime.CreateFloatingDateRule(new
DateTime(1, 1, 1, 2, 0, 0), 03, 05, DayOfWeek.Sunday);
transitionRuleEnd = TimeZoneInfo.TransitionTime.CreateFloatingDateRule(new
DateTime(1, 1, 1, 2, 0, 0), 10, 05, DayOfWeek.Sunday);
adjustment = TimeZoneInfo.AdjustmentRule.CreateAdjustmentRule(new
DateTime(1918, 1, 1), new DateTime(1919, 12, 31), delta,
transitionRuleStart, transitionRuleEnd);
adjustmentList.Add(adjustment);
// Define rule 2 (for 1942)
transitionRuleStart = TimeZoneInfo.TransitionTime.CreateFixedDateRule(new
DateTime(1, 1, 1, 2, 0, 0), 02, 09);
adjustment = TimeZoneInfo.AdjustmentRule.CreateAdjustmentRule(new
DateTime(1942, 1, 1), new DateTime(1942, 12, 31),
delta,
transitionRuleStart, transitionRuleEnd);
adjustmentList.Add(adjustment);
// Define rule 3 (for 1945)
transitionRuleStart = TimeZoneInfo.TransitionTime.CreateFixedDateRule(new
DateTime(1, 1, 1, 23, 0, 0), 08, 14);
transitionRuleEnd = TimeZoneInfo.TransitionTime.CreateFixedDateRule(new
DateTime(1, 1, 1, 2, 0, 0), 09, 30);
adjustment = TimeZoneInfo.AdjustmentRule.CreateAdjustmentRule(new
DateTime(1945, 1, 1), new DateTime(1945, 12, 31),
delta,
transitionRuleStart, transitionRuleEnd);
adjustmentList.Add(adjustment);
// Define end rule (for 1967-2006)
transitionRuleEnd = TimeZoneInfo.TransitionTime.CreateFloatingDateRule(new
DateTime(1, 1, 1, 2, 0, 0), 10, 5, DayOfWeek.Sunday);
// Define rule 4 (for 1967-73)
transitionRuleStart = TimeZoneInfo.TransitionTime.CreateFloatingDateRule(new
DateTime(1, 1, 1, 2, 0, 0), 04, 05, DayOfWeek.Sunday);
adjustment = TimeZoneInfo.AdjustmentRule.CreateAdjustmentRule(new
DateTime(1967, 1, 1), new DateTime(1973, 12, 31),
delta,
transitionRuleStart, transitionRuleEnd);
adjustmentList.Add(adjustment);
// Define rule 5 (for 1974 only)
transitionRuleStart = TimeZoneInfo.TransitionTime.CreateFixedDateRule(new
DateTime(1, 1, 1, 2, 0, 0), 01, 06);
```

```

adjustment = TimeZoneInfo.AdjustmentRule.CreateAdjustmentRule(new
DateTime(1974, 1, 1), new DateTime(1974, 12, 31),
delta,
transitionRuleStart, transitionRuleEnd);
adjustmentList.Add(adjustment);
// Define rule 6 (for 1975 only)
transitionRuleStart = TimeZoneInfo.TransitionTime.CreateFixedDateRule(new
DateTime(1, 1, 1, 2, 0, 0), 02, 23);
adjustment = TimeZoneInfo.AdjustmentRule.CreateAdjustmentRule(new
DateTime(1975, 1, 1), new DateTime(1975, 12, 31),
delta,
transitionRuleStart, transitionRuleEnd);
adjustmentList.Add(adjustment);
// Define rule 7 (1976-1986)
transitionRuleStart = TimeZoneInfo.TransitionTime.CreateFloatingDateRule(new
DateTime(1, 1, 1, 2, 0, 0), 04, 05, DayOfWeek.Sunday);
adjustment = TimeZoneInfo.AdjustmentRule.CreateAdjustmentRule(new
DateTime(1976, 1, 1), new DateTime(1986, 12, 31),
delta,
transitionRuleStart, transitionRuleEnd);
adjustmentList.Add(adjustment);
// Define rule 8 (1987-2006)
transitionRuleStart = TimeZoneInfo.TransitionTime.CreateFloatingDateRule(new
DateTime(1, 1, 1, 2, 0, 0), 04, 01, DayOfWeek.Sunday);
adjustment = TimeZoneInfo.AdjustmentRule.CreateAdjustmentRule(new
DateTime(1987, 1, 1), new DateTime(2006, 12, 31),
delta,
transitionRuleStart, transitionRuleEnd);
adjustmentList.Add(adjustment);
// Define rule 9 (2007- )
transitionRuleStart = TimeZoneInfo.TransitionTime.CreateFloatingDateRule(new
DateTime(1, 1, 1, 2, 0, 0), 03, 02, DayOfWeek.Sunday);
transitionRuleEnd = TimeZoneInfo.TransitionTime.CreateFloatingDateRule(new
DateTime(1, 1, 1, 2, 0, 0), 11, 01, DayOfWeek.Sunday);
adjustment = TimeZoneInfo.AdjustmentRule.CreateAdjustmentRule(new
DateTime(2007, 1, 1), DateTime.MaxValue.Date,
delta,
transitionRuleStart, transitionRuleEnd);
adjustmentList.Add(adjustment);

// Convert list of adjustment rules to an array
TimeZoneInfo.AdjustmentRule[] adjustments = new
TimeZoneInfo.AdjustmentRule[adjustmentList.Count];
adjustmentList.CopyTo(adjustments);

cst = TimeZoneInfo.CreateCustomTimeZone("Central Standard Time", new
TimeSpan(-6, 0, 0),
"(GMT-06:00) Central Time (US Only)", "Central Standard Time",
"Central Daylight Time", adjustments);

```

The time zone created in this example has multiple adjustment rules. Care must be taken to ensure that the effective start and end dates of any adjustment rule do not

overlap with the dates of another adjustment rule. If there is an overlap, an [InvalidTimeZoneException](#) is thrown.

For floating adjustment rules, the value 5 is passed to the `week` parameter of the [CreateFloatingDateRule](#) method to indicate that the transition occurs on the last week of a particular month.

In creating the array of [TimeZoneInfo.AdjustmentRule](#) objects to use in the [TimeZoneInfo.CreateCustomTimeZone\(String, TimeSpan, String, String, String, TimeZoneInfo+AdjustmentRule\[\]\)](#) method call, the code could initialize the array to the size required by the number of adjustments to be created for the time zone. Instead, this code example calls the [Add](#) method to add each adjustment rule to a generic `List<T>` collection of [TimeZoneInfo.AdjustmentRule](#) objects. The code then calls the [CopyTo](#) method to copy the members of this collection to the array.

The example also uses the [CreateFixedDateRule](#) method to define fixed-date adjustments. This is similar to calling the [CreateFloatingDateRule](#) method, except that it requires only the time, month, and day of the transition parameters.

The example can be tested using code such as the following:

C#

```
TimeZoneInfo est = TimeZoneInfo.FindSystemTimeZoneById("Eastern Standard Time");

DateTime pastDate1 = new DateTime(1942, 2, 11);
Console.WriteLine("Is {0} daylight saving time: {1}", pastDate1,
                  cst.IsDaylightSavingTime(pastDate1));

DateTime pastDate2 = new DateTime(1967, 10, 29, 1, 30, 00);
Console.WriteLine("Is {0} ambiguous: {1}", pastDate2,
                  cst.IsAmbiguousTime(pastDate2));

DateTime pastDate3 = new DateTime(1974, 1, 7, 2, 59, 00);
Console.WriteLine("{0} {1} is {2} {3}", pastDate3,
                  est.IsDaylightSavingTime(pastDate3) ?
                      est.DaylightName : est.StandardName,
                  TimeZoneInfo.ConvertTime(pastDate3, est, cst),

                  cst.IsDaylightSavingTime(TimeZoneInfo.ConvertTime(pastDate3, est, cst)) ?
                      cst.DaylightName : cst.StandardName);
// 
// This code produces the following output to the console:
//
//      Is 2/11/1942 12:00:00 AM daylight saving time: True
//      Is 10/29/1967 1:30:00 AM ambiguous: True
```

```
// 1/7/1974 2:59:00 AM Eastern Standard Time is 1/7/1974 2:59:00 AM  
Central Daylight Time
```

Compiling the code

This example requires:

- That the following namespaces be imported:

C#

```
using System.Collections.Generic;  
using System.Collections.ObjectModel;
```

See also

- [Dates, times, and time zones](#)
- [Time zone overview](#)
- [How to: Create time zones without adjustment rules](#)

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

Finding the time zones defined on a local system

Article • 09/15/2021

The [TimeZoneInfo](#) class does not expose a public constructor. As a result, the `new` keyword cannot be used to create a new [TimeZoneInfo](#) object. Instead, [TimeZoneInfo](#) objects are instantiated either by retrieving information on predefined time zones from the registry or by creating a custom time zone. This topic discusses instantiating a time zone from data stored in the registry. In addition, `static` (`shared` in Visual Basic) properties of the [TimeZoneInfo](#) class provide access to Coordinated Universal Time (UTC) and the local time zone.

ⓘ Note

For time zones that are not defined in the registry, you can create custom time zones by calling the overloads of the [CreateCustomTimeZone](#) method. Creating a custom time zone is discussed in the [How to: Create time zones without adjustment rules](#) and [How to: Create time zones with adjustment rules](#) topics. In addition, you can instantiate a [TimeZoneInfo](#) object by restoring it from a serialized string with the [FromSerializedString](#) method. Serializing and deserializing a [TimeZoneInfo](#) object is discussed in the [How to: Save time zones to an embedded resource](#) and [How to: Restore Time Zones from an Embedded Resource](#) topics.

Accessing individual time zones

The [TimeZoneInfo](#) class provides two predefined time zone objects that represent the UTC time and the local time zone. They are available from the [Utc](#) and [Local](#) properties, respectively. For instructions on accessing the UTC or local time zones, see [How to: Access the predefined UTC and local time zone objects](#).

You can also instantiate a [TimeZoneInfo](#) object that represents any time zone defined in the registry. For instructions on instantiating a specific time zone object, see [How to: Instantiate a TimeZoneInfo object](#).

Time zone identifiers

The time zone identifier is a key field that uniquely identifies the time zone. While most keys are relatively short, the time zone identifier is comparatively long. In most cases, its

value corresponds to the [TimeZoneInfo.StandardName](#) property, which is used to provide the name of the time zone's standard time. However, there are exceptions. The best way to make sure that you supply a valid identifier is to enumerate the time zones available on your system and note their associated identifiers.

See also

- [Dates, times, and time zones](#)
- [How to: Access the predefined UTC and local time zone objects](#)
- [How to: Instantiate a TimeZoneInfo object](#)
- [Converting times between time zones](#)

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback [here](#).

 [Open a documentation issue](#)

 [Provide product feedback](#)

How to: Enumerate time zones present on a computer

Article • 09/15/2021

Successfully working with a designated time zone requires that information about that time zone be available to the system. The Windows XP and Windows Vista operating systems store this information in the registry. However, although the total number of time zones that exist throughout the world is large, the registry contains information about only a subset of them. In addition, the registry itself is a dynamic structure whose contents are subject to both deliberate and accidental change. As a result, an application cannot always assume that a particular time zone is defined and available on a system. The first step for many applications that use time zone information applications is to determine whether required time zones are available on the local system, or to give the user a list of time zones from which to select. This requires that an application enumerate the time zones defined on a local system.

ⓘ Note

If an application relies on the presence of a particular time zone that may not be defined on a local system, the application can ensure its presence by serializing and deserializing information about the time zone. The time zone can then be added to a list control so that the application user can select it. For details, see [How to: Save Time Zones to an Embedded Resource](#) and [How to: Restore time zones from an embedded resource](#).

To enumerate the time zones present on the local system

1. Call the [TimeZoneInfo.GetSystemTimeZones](#) method. The method returns a generic [ReadOnlyCollection<T>](#) collection of [TimeZoneInfo](#) objects. The entries in the collection are sorted by their [DisplayName](#) property. For example:

C#

```
ReadOnlyCollection<TimeZoneInfo> tzCollection;  
tzCollection = TimeZoneInfo.GetSystemTimeZones();
```

2. Enumerate the individual [TimeZoneInfo](#) objects in the collection by using a [foreach](#) loop (in C#) or a [For Each...Next](#) loop (in Visual Basic), and perform any necessary processing on each object. For example, the following code enumerates

the `ReadOnlyCollection<T>` collection of `TimeZoneInfo` objects returned in step 1 and lists the display name of each time zone on the console.

```
C#  
  
foreach (TimeZoneInfo timeZone in tzCollection)  
    Console.WriteLine(" {0}: {1}", timeZone.Id, timeZone.DisplayName);
```

To present the user with a list of time zones present on the local system

1. Call the `TimeZoneInfo.GetSystemTimeZones` method. The method returns a generic `ReadOnlyCollection<T>` collection of `TimeZoneInfo` objects.
2. Assign the collection returned in step 1 to the `DataSource` property of a Windows forms or ASP.NET list control.
3. Retrieve the `TimeZoneInfo` object that the user has selected.

The example provides an illustration for a Windows application.

Example

The example starts a Windows application that displays the time zones defined on a system in a list box. The example then displays a dialog box that contains the value of the `DisplayName` property of the time zone object selected by the user.

```
C#  
  
private void Form1_Load(object sender, EventArgs e)  
{  
    ReadOnlyCollection<TimeZoneInfo> tzCollection;  
    tzCollection = TimeZoneInfo.GetSystemTimeZones();  
    this.timeZoneList.DataSource = tzCollection;  
}  
  
private void OkButton_Click(object sender, EventArgs e)  
{  
    TimeZoneInfo selectedTimeZone = (TimeZoneInfo)  
this.timeZoneList.SelectedItem;  
    MessageBox.Show("You selected the " + selectedTimeZone.ToString() + "  
time zone.");  
}
```

Most list controls (such as the `System.Windows.Forms.ListBox` or `System.Web.UI.WebControls.BulletedList` control) allow you to assign a collection of object variables to their `DataSource` property as long as that collection implements the `IEnumerable` interface. (The generic `ReadOnlyCollection<T>` class does this.) To display an individual object in the collection, the control calls that object's `ToString` method to extract the string that is used to represent the object. In the case of `TimeZoneInfo` objects, the `ToString` method returns the `TimeZoneInfo` object's display name (the value of its `DisplayName` property).

ⓘ Note

Because list controls call an object's `ToString` method, you can assign a collection of `TimeZoneInfo` objects to the control, have the control display a meaningful name for each object, and retrieve the `TimeZoneInfo` object that the user has selected. This eliminates the need to extract a string for each object in the collection, assign the string to a collection that is in turn assigned to the control's `DataSource` property, retrieve the string the user has selected, and then use this string to extract the object that it describes.

Compiling the code

This example requires:

- That the following namespaces be imported:

`System` (in C# code)

`System.Collections.ObjectModel`

See also

- [Dates, times, and time zones](#)
- [How to: Save time zones to an embedded resource](#)
- [How to: Restore time zones from an embedded resource](#)

 Collaborate with us on
GitHub



.NET feedback

The .NET documentation is open source. Provide feedback [here](#).

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

 [Open a documentation issue](#)

 [Provide product feedback](#)

How to: Access the predefined UTC and local time zone objects

Article • 09/15/2021

The [TimeZoneInfo](#) class provides two properties, [Utc](#) and [Local](#), that give your code access to predefined time zone objects. This topic discusses how to access the [TimeZoneInfo](#) objects returned by those properties.

To access the Coordinated Universal Time (UTC) [TimeZoneInfo](#) object

1. Use the `static` (`Shared` in Visual Basic) [TimeZoneInfo.Utc](#) property to access Coordinated Universal Time.
2. Rather than assigning the [TimeZoneInfo](#) object returned by the property to an object variable, continue to access Coordinated Universal Time through the [TimeZoneInfo.Utc](#) property.

To access the local time zone

1. Use the `static` (`Shared` in Visual Basic) [TimeZoneInfo.Local](#) property to access the local system time zone.
2. Rather than assigning the [TimeZoneInfo](#) object returned by the property to an object variable, continue to access the local time zone through the [TimeZoneInfo.Local](#) property.

Example

The following code uses the [TimeZoneInfo.Local](#) and [TimeZoneInfo.Utc](#) properties to convert a time from the U.S. and Canadian Eastern Standard time zone, as well as to display the time zone name to the console.

C#

```
// Create Eastern Standard Time value and TimeZoneInfo object
DateTime estTime = new DateTime(2007, 1, 1, 00, 00, 00);
string timeZoneName = "Eastern Standard Time";
try
{
    TimeZoneInfo est = TimeZoneInfo.FindSystemTimeZoneById(timeZoneName);
```

```

// Convert EST to local time
DateTime localTime = TimeZoneInfo.ConvertTime(estTime, est,
TimeZoneInfo.Local);
Console.WriteLine("At {0} {1}, the local time is {2} {3}.",
    estTime,
    est,
    localTime,
    TimeZoneInfo.Local.IsDaylightSavingTime(localTime) ?
        TimeZoneInfo.Local.DaylightName :
        TimeZoneInfo.Local.StandardName);

// Convert EST to UTC
DateTime utcTime = TimeZoneInfo.ConvertTime(estTime, est,
TimeZoneInfo.Utc);
Console.WriteLine("At {0} {1}, the time is {2} {3}.",
    estTime,
    est,
    utcTime,
    TimeZoneInfo.Utc.StandardName);
}

catch (TimeZoneNotFoundException)
{
    Console.WriteLine("The {0} zone cannot be found in the registry.",
        timeZoneName);
}
catch (InvalidTimeZoneException)
{
    Console.WriteLine("The registry contains invalid data for the {0} zone.",
        timeZoneName);
}

// The example produces the following output to the console:
// At 1/1/2007 12:00:00 AM (UTC-05:00) Eastern Time (US & Canada), the
// local time is 1/1/2007 12:00:00 AM Eastern Standard Time.
// At 1/1/2007 12:00:00 AM (UTC-05:00) Eastern Time (US & Canada), the
// time is 1/1/2007 5:00:00 AM UTC.

```

You should always access the local time zone through the [TimeZoneInfo.Local](#) property rather than assigning the local time zone to a [TimeZoneInfo](#) object variable. Similarly, you should always access Coordinated Universal Time through the [TimeZoneInfo.Utc](#) property rather than assigning the UTC zone to a [TimeZoneInfo](#) object variable. This prevents the [TimeZoneInfo](#) object variable from being invalidated by a call to the [TimeZoneInfo.ClearCachedData](#) method.

Compiling the code

This example requires:

- That the `System` namespace be imported with the `using` statement (required in C# code).

See also

- [Dates, times, and time zones](#)
- [Finding the time zones defined on a local system](#)
- [How to: Instantiate a TimeZoneInfo object](#)

 [Collaborate with us on GitHub](#)

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

How to: Instantiate a TimeZoneInfo object

Article • 09/15/2021

The most common way to instantiate a [TimeZoneInfo](#) object is to retrieve information about it from the registry. This topic discusses how to instantiate a [TimeZoneInfo](#) object from the local system registry.

To instantiate a TimeZoneInfo object

1. Declare a [TimeZoneInfo](#) object.
2. Call the `static` (`Shared` in Visual Basic) [TimeZoneInfo.FindSystemTimeZoneById](#) method.
3. Handle any exceptions thrown by the method, particularly the [TimeZoneNotFoundException](#) that is thrown if the time zone is not defined in the registry.

Example

The following code retrieves a [TimeZoneInfo](#) object that represents the Eastern Standard Time zone and displays the Eastern Standard time that corresponds to the local time.

C#

```
DateTime timeNow = DateTime.Now;
try
{
    TimeZoneInfo easternZone = TimeZoneInfo.FindSystemTimeZoneById("Eastern
Standard Time");
    DateTime easternTimeNow = TimeZoneInfo.ConvertTime(timeNow,
TimeZoneInfo.Local,
                                                    easternZone);
    Console.WriteLine("{0} {1} corresponds to {2} {3}.",
                      timeNow,
                      TimeZoneInfo.Local.IsDaylightSavingTime(timeNow) ?
                        TimeZoneInfo.Local.DaylightName :
                        TimeZoneInfo.Local.StandardName,
                      easternTimeNow,
                      easternZone.IsDaylightSavingTime(easternTimeNow) ?
                        easternZone.DaylightName :
                        easternZone.StandardName);
}
```

```

// Handle exception
//
// As an alternative to simply displaying an error message, an alternate
Eastern
// Standard Time TimeZoneInfo object could be instantiated here either by
restoring
// it from a serialized string or by providing the necessary data to the
// CreateCustomTimeZone method.
catch (TimeZoneNotFoundException)
{
    Console.WriteLine("The Eastern Standard Time Zone cannot be found on the
local system.");
}
catch (InvalidTimeZoneException)
{
    Console.WriteLine("The Eastern Standard Time Zone contains invalid or
missing data.");
}
catch (SecurityException)
{
    Console.WriteLine("The application lacks permission to read time zone
information from the registry.");
}
catch (OutOfMemoryException)
{
    Console.WriteLine("Not enough memory is available to load information on
the Eastern Standard Time zone.");
}
// If we weren't passing FindSystemTimeZoneById a literal string, we also
// would handle an ArgumentNullException.

```

The `TimeZoneInfo.FindSystemTimeZoneById` method's single parameter is the identifier of the time zone that you want to retrieve, which corresponds to the object's `TimeZoneInfo.Id` property. The time zone identifier is a key field that uniquely identifies the time zone. While most keys are relatively short, the time zone identifier is comparatively long. In most cases, its value corresponds to the `StandardName` property of a `TimeZoneInfo` object, which is used to provide the name of the time zone's standard time. However, there are exceptions. The best way to make sure that you supply a valid identifier is to enumerate the time zones available on your system and note the identifiers of the time zones present on them. For an illustration, see [How to: Enumerate time zones present on a computer](#). The [Finding the time zones defined on a local system](#) topic also contains a list of selected time zone identifiers.

If the time zone is found, the method returns its `TimeZoneInfo` object. If the time zone is not found, the method throws a `TimeZoneNotFoundException`. If the time zone is found but its data is corrupted or incomplete, the method throws an `InvalidTimeZoneException`.

If your application relies on a time zone that must be present, you should first call the [FindSystemTimeZoneById](#) method to retrieve the time zone information from the registry. If the method call fails, your exception handler should then either create a new instance of the time zone or re-create it by deserializing a serialized [TimeZoneInfo](#) object. See [How to: Restore time zones from an embedded resource](#) for an example.

See also

- [Dates, times, and time zones](#)
- [Finding the time zones defined on a local system](#)
- [How to: Access the predefined UTC and local time zone objects](#)

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

Saving and restoring time zones

Article • 09/15/2021

The [TimeZoneInfo](#) class relies on the registry to retrieve predefined time zone data. However, the registry is a dynamic structure. Additionally, the time zone information that the registry contains is used by the operating system primarily to handle time adjustments and conversions for the current year. This has two major implications for applications that rely on accurate time zone data:

- A time zone that is required by an application may not be defined in the registry, or it may have been renamed or removed from the registry.
- A time zone that is defined in the registry may lack information about the particular adjustment rules that are necessary for historical time zone conversions.

The [TimeZoneInfo](#) class addresses these limitations through its support for serialization (saving) and deserialization (restoring) of time zone data.

Time zone serialization and deserialization

Saving and restoring a time zone by serializing and deserializing time zone data involves just two method calls:

- You can serialize a [TimeZoneInfo](#) object by calling that object's [ToSerializedString](#) method. The method takes no parameters and returns a string that contains time zone information.
- You can deserialize a [TimeZoneInfo](#) object from a serialized string by passing that string to the `static (Shared in Visual Basic)` [TimeZoneInfo.FromSerializedString](#) method.

Serialization and deserialization scenarios

The ability to save (or serialize) a [TimeZoneInfo](#) object to a string and to restore (or deserialize) it for later use increases both the utility and the flexibility of the [TimeZoneInfo](#) class. This section examines some of the situations in which serialization and deserialization are most useful.

Serializing and deserializing time zone data in an application

A serialized time zone can be restored from a string when it is needed. An application might do this if the time zone retrieved from the registry is unable to correctly convert a date and time within a particular date range. For example, time zone data in the Windows XP registry supports a single adjustment rule, while time zones defined in the Windows Vista registry typically provide information about two adjustment rules. This means that historical time conversions may be inaccurate. Serialization and deserialization of time zone data can handle this limitation.

In the following example, a custom `TimeZoneInfo` class that has no adjustment rules is defined to represent the U.S. Eastern Standard Time zone from 1883 to 1917, before the introduction of daylight saving time in the United States. The custom time zone is serialized in a variable that has global scope. The time zone conversion method, `ConvertUtcTime`, is passed Coordinated Universal Time (UTC) times to convert. If the date and time occurs in 1917 or earlier, the custom Eastern Standard Time zone is restored from a serialized string and replaces the time zone retrieved from the registry.

C#

```
using System;

public class TimeZoneSerialization
{
    static string serializedEst;

    public static void Main()
    {
        // Retrieve Eastern Standard Time zone from registry
        try
        {
            TimeZoneSerialization tzs = new TimeZoneSerialization();
            TimeZoneInfo est = TimeZoneInfo.FindSystemTimeZoneById("Eastern
Standard Time");
            // Create custom Eastern Time Zone for historical (pre-1918)
            // conversions
            CreateTimeZone();
            // Call conversion function with one current and one pre-1918 date
            // and time
            Console.WriteLine(ConvertUtcTime(DateTime.UtcNow, est));
            Console.WriteLine(ConvertUtcTime(new DateTime(1900, 11, 15, 9, 32,
00, DateTimeKind.Utc), est));
        }
        catch (TimeZoneNotFoundException)
        {
            Console.WriteLine("The Eastern Standard Time zone is not in the
registry.");
        }
        catch (InvalidTimeZoneException)
        {
            Console.WriteLine("Data on the Eastern Standard Time Zone in the
registry is corrupted.");
        }
    }
}
```

```

        }

    }

    private static void CreateTimeZone()
    {
        // Create a simple Eastern Standard time zone
        // without adjustment rules for 1883-1918
        TimeZoneInfo earlyEstZone = TimeZoneInfo.CreateCustomTimeZone("Eastern
Standard Time",
            new TimeSpan(-5, 0, 0),
            "(GMT-05:00) Eastern Time (United
States)",
            "Eastern Standard Time");
        serializedEst = earlyEstZone.ToSerializedString();
    }

    private static DateTime ConvertUtcTime(DateTime utcDate, TimeZoneInfo tz)
{
    // Use time zone object from registry
    if (utcDate.Year > 1917)
    {
        return TimeZoneInfo.ConvertTimeFromUtc(utcDate, tz);
    }
    // Handle dates before introduction of DST
    else
    {
        // Restore serialized time zone object
        tz = TimeZoneInfo.FromSerializedString(serializedEst);
        return TimeZoneInfo.ConvertTimeFromUtc(utcDate, tz);
    }
}
}

```

Handling time zone exceptions

Because the registry is a dynamic structure, its contents are subject to accidental or deliberate modification. This means that a time zone that should be defined in the registry and that is required for an application to execute successfully may be absent. Without support for time zone serialization and deserialization, you have little choice but to handle the resulting [TimeZoneNotFoundException](#) by ending the application. However, by using time zone serialization and deserialization, you can handle an unexpected [TimeZoneNotFoundException](#) by restoring the required time zone from a serialized string, and the application will continue to run.

The following example creates and serializes a custom Central Standard Time zone. It then tries to retrieve the Central Standard Time zone from the registry. If the retrieval operation throws either a [TimeZoneNotFoundException](#) or an [InvalidTimeZoneException](#), the exception handler deserializes the time zone.

C#

```
using System;
using System.Collections.Generic;

public class TimeZoneApplication
{
    // Define collection of custom time zones
    private Dictionary<string, string> customTimeZones = new
Dictionary<string, string>();
    private TimeZoneInfo cst;

    public TimeZoneApplication()
    {
        // Create custom Central Standard Time
        //
        // Declare necessary TimeZoneInfo.AdjustmentRule objects for time zone
        TimeZoneInfo customTimeZone;
        TimeSpan delta = new TimeSpan(1, 0, 0);
        TimeZoneInfo.AdjustmentRule adjustment;
        List<TimeZoneInfo.AdjustmentRule> adjustmentList = new
List<TimeZoneInfo.AdjustmentRule>();
        // Declare transition time variables to hold transition time
        information
        TimeZoneInfo.TransitionTime transitionRuleStart, transitionRuleEnd;

        // Define end rule (for 1976-2006)
        transitionRuleEnd =
TimeZoneInfo.TransitionTime.CreateFloatingDateRule(new DateTime(1, 1, 1, 2,
0, 0), 10, 5, DayOfWeek.Sunday);
        // Define rule (1976-1986)
        transitionRuleStart =
TimeZoneInfo.TransitionTime.CreateFloatingDateRule(new DateTime(1, 1, 1, 2,
0, 0), 04, 05, DayOfWeek.Sunday);
        adjustment = TimeZoneInfo.AdjustmentRule.CreateAdjustmentRule(new
DateTime(1976, 1, 1), new DateTime(1986, 12, 31), delta,
transitionRuleStart, transitionRuleEnd);
        adjustmentList.Add(adjustment);
        // Define rule (1987-2006)
        transitionRuleStart =
TimeZoneInfo.TransitionTime.CreateFloatingDateRule(new DateTime(1, 1, 1, 2,
0, 0), 04, 01, DayOfWeek.Sunday);
        adjustment = TimeZoneInfo.AdjustmentRule.CreateAdjustmentRule(new
DateTime(1987, 1, 1), new DateTime(2006, 12, 31), delta,
transitionRuleStart, transitionRuleEnd);
        adjustmentList.Add(adjustment);
        // Define rule (2007- )
        transitionRuleStart =
TimeZoneInfo.TransitionTime.CreateFloatingDateRule(new DateTime(1, 1, 1, 2,
0, 0), 03, 02, DayOfWeek.Sunday);
        transitionRuleEnd =
TimeZoneInfo.TransitionTime.CreateFloatingDateRule(new DateTime(1, 1, 1, 2,
0, 0), 11, 01, DayOfWeek.Sunday);
        adjustment = TimeZoneInfo.AdjustmentRule.CreateAdjustmentRule(new
```

```

        DateTime(2007, 01, 01), DateTime.MaxValue.Date, delta, transitionRuleStart,
        transitionRuleEnd);
        adjustmentList.Add(adjustment);

        // Create custom U.S. Central Standard Time zone
        customTimeZone = TimeZoneInfo.CreateCustomTimeZone("Central Standard
Time",
            new TimeSpan(-6, 0, 0),
            "(GMT-06:00) Central Time (US Only)", "Central
Standard Time",
            "Central Daylight Time", adjustmentList.ToArray());
        // Add time zone to collection
        customTimeZones.Add(customTimeZone.Id,
        customTimeZone.ToSerializedString());

        // Create any other required time zones
    }

    public static void Main()
{
    TimeZoneApplication tza = new TimeZoneApplication();
    tza.AppEntryPoint();
}

private void AppEntryPoint()
{
    try
    {
        cst = TimeZoneInfo.FindSystemTimeZoneById("Central Standard Time");
    }
    catch (TimeZoneNotFoundException)
    {
        if (customTimeZones.ContainsKey("Central Standard Time"))
            HandleTimeZoneException("Central Standard Time");
    }
    catch (InvalidTimeZoneException)
    {
        if (customTimeZones.ContainsKey("Central Standard Time"))
            HandleTimeZoneException("Central Standard Time");
    }
    if (cst == null)
    {
        Console.WriteLine("Unable to load Central Standard Time zone.");
        return;
    }
    DateTime currentTime = DateTime.Now;
    Console.WriteLine("The current {0} time is {1}.",
        TimeZoneInfo.Local.IsDaylightSavingTime(currentTime)
?
                TimeZoneInfo.Local.StandardName :
                TimeZoneInfo.Local.DaylightName,
                currentTime.ToString("f"));
    Console.WriteLine("The current {0} time is {1}.",
        cst.IsDaylightSavingTime(currentTime) ?
                cst.StandardName :

```

```
        cst.DaylightName,
        TimeZoneInfo.ConvertTime(currentTime,
TimeZoneInfo.Local, cst).ToString("f"));
    }

    private void HandleTimeZoneException(string timeZoneName)
{
    string tzString = customTimeZones[timeZoneName];
    cst = TimeZoneInfo.FromSerializedString(tzString);
}
}
```

Storing a serialized string and restoring it when needed

The previous examples have stored time zone information to a string variable and restored it when needed. However, the string that contains serialized time zone information can itself be stored in some storage medium, such as an external file, a resource file embedded in the application, or the registry. (Note that information about custom time zones should be stored apart from the system's time zone keys in the registry.)

Storing a serialized time zone string in this manner also separates the time zone creation routine from the application itself. For example, a time zone creation routine can execute and create a data file that contains historical time zone information that an application can use. The data file can be then be installed with the application, and it can be opened and one or more of its time zones can be deserialized when the application requires them.

For an example that uses an embedded resource to store serialized time zone data, see [How to: Save time zones to an embedded resource](#) and [How to: Restore time zones from an embedded resource](#).

See also

- [Dates, times, and time zones](#)

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

more information, see [our contributor guide](#).



[Provide product feedback](#)

How to: Save time zones to an embedded resource

Article • 09/15/2021

A time zone-aware application often requires the presence of a particular time zone. However, because the availability of individual [TimeZoneInfo](#) objects depends on information stored in the local system's registry, even customarily available time zones may be absent. In addition, information about custom time zones instantiated by using the [CreateCustomTimeZone](#) method is not stored with other time zone information in the registry. To ensure that these time zones are available when they are needed, you can save them by serializing them, and later restore them by deserializing them.

Typically, serializing a [TimeZoneInfo](#) object occurs apart from the time zone-aware application. Depending on the data store used to hold serialized [TimeZoneInfo](#) objects, time zone data may be serialized as part of a setup or installation routine (for example, when the data is stored in an application key of the registry), or as part of a utility routine that runs before the final application is compiled (for example, when the serialized data is stored in a .NET XML resource (.resx) file).

In addition to a resource file that is compiled with the application, several other data stores can be used for time zone information. These include the following:

- The registry. Note that an application should use the subkeys of its own application key to store custom time zone data rather than using the subkeys of HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Time Zones.
- Configuration files.
- Other system files.

To save a time zone by serializing it to a .resx file

1. Retrieve an existing time zone or create a new time zone.

To retrieve an existing time zone, see [How to: Access the predefined UTC and local time zone objects](#) and [How to: Instantiate a TimeZoneInfo object](#).

To create a new time zone, call one of the overloads of the [CreateCustomTimeZone](#) method. For more information, see [How to: Create time zones without adjustment rules](#) and [How to: Create time zones with adjustment rules](#).

2. Call the [ToSerializedString](#) method to create a string that contains the time zone's data.
3. Instantiate a [StreamWriter](#) object by providing the name and optionally the path of the .resx file to the [StreamWriter](#) class constructor.
4. Instantiate a [ResXResourceWriter](#) object by passing the [StreamWriter](#) object to the [ResXResourceWriter](#) class constructor.
5. Pass the time zone's serialized string to the [ResXResourceWriter.AddResource](#) method.
6. Call the [ResXResourceWriter.Generate](#) method.
7. Call the [ResXResourceWriter.Close](#) method.
8. Close the [StreamWriter](#) object by calling its [Close](#) method.
9. Add the generated .resx file to the application's Visual Studio project.
10. Using the [Properties](#) window in Visual Studio, make sure that the .resx file's **Build Action** property is set to **Embedded Resource**.

Example

The following example serializes a [TimeZoneInfo](#) object that represents Central Standard Time and a [TimeZoneInfo](#) object that represents the Palmer Station, Antarctica time to a .NET XML resource file that is named SerializedTimeZones.resx. Central Standard Time is typically defined in the registry; Palmer Station, Antarctica is a custom time zone.

C#

```
TimeZoneSerialization()
{
    TextWriter writeStream;
    Dictionary<string, string> resources = new Dictionary<string, string>();
    // Determine if .resx file exists
    if (File.Exists(resxName))
    {
        // Open reader
        TextReader readStream = new StreamReader(resxName);
        ResXResourceReader resReader = new ResXResourceReader(readStream);
        foreach (DictionaryEntry item in resReader)
        {
            if (! (((string) item.Key) == "CentralStandardTime" ||
                   ((string) item.Key) == "PalmerStandardTime" ))
                resources.Add((string)item.Key, (string) item.Value);
        }
    }
}
```

```

        readStream.Close();
        // Delete file, since write method creates duplicate xml headers
        File.Delete(resxName);
    }

    // Open stream to write to .resx file
    try
    {
        writeStream = new StreamWriter(resxName, true);
    }
    catch (FileNotFoundException e)
    {
        // Handle failure to find file
        Console.WriteLine("{0}: The file {1} could not be found.",
e.GetType().Name, resxName);
        return;
    }

    // Get resource writer
    ResXResourceWriter resWriter = new ResXResourceWriter(writeStream);

    // Add resources from existing file
    foreach (KeyValuePair<string, string> item in resources)
    {
        resWriter.AddResource(item.Key, item.Value);
    }

    // Serialize Central Standard Time
    try
    {
        TimeZoneInfo cst = TimeZoneInfo.FindSystemTimeZoneById("Central
Standard Time");
        resWriter.AddResource(cst.Id.Replace(" ", string.Empty),
cst.ToSerializedString());
    }
    catch (TimeZoneNotFoundException)
    {
        Console.WriteLine("The Central Standard Time zone could not be
found.");
    }

    // Create time zone for Palmer, Antarctica
    //
    // Define transition times to/from DST
    TimeZoneInfo.TransitionTime startTransition =
TimeZoneInfo.TransitionTime.CreateFloatingDateRule(new DateTime(1, 1, 1, 4,
0, 0),

10, 2, DayOfWeek.Sunday);
    TimeZoneInfo.TransitionTime endTransition =
TimeZoneInfo.TransitionTime.CreateFloatingDateRule(new DateTime(1, 1, 1, 3,
0, 0),

3, 2, DayOfWeek.Sunday);
    // Define adjustment rule

```

```

        TimeSpan delta = new TimeSpan(1, 0, 0);
        TimeZoneInfo.AdjustmentRule adjustment =
            TimeZoneInfo.AdjustmentRule.CreateAdjustmentRule(new DateTime(1999, 10, 1),
                DateTime.MaxValue.Date, delta,
                startTransition, endTransition);
        // Create array for adjustment rules
        TimeZoneInfo.AdjustmentRule[] adjustments = {adjustment};
        // Define other custom time zone arguments
        string DisplayName = "(GMT-04:00) Antarctica/Palmer Time";
        string standardName = "Palmer Standard Time";
        string daylightName = "Palmer Daylight Time";
        TimeSpan offset = new TimeSpan(-4, 0, 0);
        TimeZoneInfo palmer = TimeZoneInfo.CreateCustomTimeZone(standardName,
        offset, DisplayName, standardName, daylightName, adjustments);
        resWriter.AddResource(palmer.Id.Replace(" ", String.Empty),
        palmer.ToSerializedString());

        // Save changes to .resx file
        resWriter.Generate();
        resWriter.Close();
        writeStream.Close();
    }
}

```

This example serializes [TimeZoneInfo](#) objects so that they are available in a resource file at compile time.

Because the [ResXResourceWriter.Generate](#) method adds complete header information to a .NET XML resource file, it cannot be used to add resources to an existing file. The example handles this by checking for the SerializedTimeZones.resx file and, if it exists, storing all of its resources other than the two serialized time zones to a generic [Dictionary<TKey, TValue>](#) object. The existing file is then deleted and the existing resources are added to a new SerializedTimeZones.resx file. The serialized time zone data is also added to this file.

The key (or [Name](#)) fields of resources should not contain embedded spaces. The [Replace\(String, String\)](#) method is called to remove all embedded spaces in the time zone identifiers before they are assigned to the resource file.

Compiling the code

This example requires:

- That a reference to [System.Windows.Forms.dll](#) and [System.Core.dll](#) be added to the project.
- That the following namespaces be imported:

C#

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Globalization;
using System.IO;
using System.Reflection;
using System.Resources;
using System.Windows.Forms;
```

See also

- [Dates, times, and time zones](#)
- [Time zone overview](#)
- [How to: Restore time zones from an embedded resource](#)

 [Collaborate with us on GitHub](#)

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

How to: Restore time zones from an embedded resource

Article • 09/15/2021

This topic describes how to restore time zones that have been saved in a resource file. For information and instructions about saving time zones, see [How to: Save time zones to an embedded resource](#).

To deserialize a `TimeZoneInfo` object from an embedded resource

1. If the time zone to be retrieved is not a custom time zone, try to instantiate it by using the [FindSystemTimeZoneByIId](#) method.
2. Instantiate a [ResourceManager](#) object by passing the fully qualified name of the embedded resource file and a reference to the assembly that contains the resource file.

If you cannot determine the fully qualified name of the embedded resource file, use the [Ilasm.exe \(IL Disassembler\)](#) to examine the assembly's manifest. An `.mresource` entry identifies the resource. In the example, the resource's fully qualified name is `SerializeTimeZoneData.SerializedTimeZones`.

If the resource file is embedded in the same assembly that contains the time zone instantiation code, you can retrieve a reference to it by calling the `static` (Shared in Visual Basic) [GetExecutingAssembly](#) method.

3. If the call to the [FindSystemTimeZoneByIId](#) method fails, or if a custom time zone is to be instantiated, retrieve a string that contains the serialized time zone by calling the [ResourceManager.GetString](#) method.
4. Deserialize the time zone data by calling the [FromSerializedString](#) method.

Example

The following example deserializes a `TimeZoneInfo` object stored in an embedded .NET XML resource file.

C#

```

private void DeserializeTimeZones()
{
    TimeZoneInfo cst, palmer;
    string timeZoneString;
    ResourceManager resMgr = new
    ResourceManager("SerializeTimeZoneData.SerializedTimeZones",
    this.GetType().Assembly);

    // Attempt to retrieve time zone from system
    try
    {
        cst = TimeZoneInfo.FindSystemTimeZoneById("Central Standard Time");
    }
    catch (TimeZoneNotFoundException)
    {
        // Time zone not in system; retrieve from resource
        timeZoneString = resMgr.GetString("CentralStandardTime");
        if (!String.IsNullOrEmpty(timeZoneString))
        {
            cst = TimeZoneInfo.FromSerializedString(timeZoneString);
        }
        else
        {
            MessageBox.Show("Unable to create Central Standard Time Zone.
Application must exit.", "Application Error");
            return;
        }
    }
    // Retrieve custom time zone
    try
    {
        timeZoneString = resMgr.GetString("PalmerStandardTime");
        palmer = TimeZoneInfo.FromSerializedString(timeZoneString);
    }
    catch (MissingManifestResourceException)
    {
        MessageBox.Show("Unable to retrieve the Palmer Standard Time Zone from
the resource file. Application must exit.");
        return;
    }
}

```

This code illustrates exception handling to ensure that a [TimeZoneInfo](#) object required by the application is present. It first tries to instantiate a [TimeZoneInfo](#) object by retrieving it from the registry using the [FindSystemTimeZoneByld](#) method. If the time zone cannot be instantiated, the code retrieves it from the embedded resource file.

Because data for custom time zones (time zones instantiated by using the [CreateCustomTimeZone](#) method) are not stored in the registry, the code does not call the [FindSystemTimeZoneByld](#) to instantiate the time zone for Palmer, Antarctica.

Instead, it immediately looks to the embedded resource file to retrieve a string that contains the time zone's data before it calls the [FromSerializedString](#) method.

Compiling the code

This example requires:

- That a reference to System.Windows.Forms.dll and System.Core.dll be added to the project.
- That the following namespaces be imported:

C#

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Globalization;
using System.IO;
using System.Reflection;
using System.Resources;
using System.Windows.Forms;
```

See also

- [Dates, times, and time zones](#)
- [Time zone overview](#)
- [How to: Save time zones to an embedded resource](#)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback [here](#).

 [Open a documentation issue](#)

 [Provide product feedback](#)

System.DateTime struct

Article • 01/08/2024

This article provides supplementary remarks to the reference documentation for this API.

ⓘ Important

Eras in the Japanese calendars are based on the emperor's reign and are therefore expected to change. For example, May 1, 2019 marked the beginning of the Reiwa era in the [JapaneseCalendar](#) and [JapaneseLunisolarCalendar](#). Such a change of era affects all applications that use these calendars. For more information and to determine whether your applications are affected, see [Handling a new era in the Japanese calendar in .NET](#). For information on testing your applications on Windows systems to ensure their readiness for the era change, see [Prepare your application for the Japanese era change](#). For features in .NET that support calendars with multiple eras and for best practices when working with calendars that support multiple eras, see [Working with eras](#).

Overview

The [DateTime](#) value type represents dates and times with values ranging from 00:00:00 (midnight), January 1, 0001 Anno Domini (Common Era) through 11:59:59 P.M., December 31, 9999 A.D. (C.E.) in the Gregorian calendar.

Time values are measured in 100-nanosecond units called ticks. A particular date is the number of ticks since 12:00 midnight, January 1, 0001 A.D. (C.E.) in the [GregorianCalendar](#) calendar. The number excludes ticks that would be added by leap seconds. For example, a ticks value of 31241376000000000L represents the date Friday, January 01, 0100 12:00:00 midnight. A [DateTime](#) value is always expressed in the context of an explicit or default calendar.

ⓘ Note

If you're working with a ticks value that you want to convert to some other time interval, such as minutes or seconds, you should use the [TimeSpan.TicksPerDay](#), [TimeSpan.TicksPerHour](#), [TimeSpan.TicksPerMinute](#), [TimeSpan.TicksPerSecond](#), or [TimeSpan.TicksPerMillisecond](#) constant to perform the conversion. For example, to add the number of seconds represented by a specified number of ticks to the

Second component of a `DateTime` value, you can use the expression

```
dateValue.Second + nTicks/Timespan.TicksPerSecond.
```

You can view the source for the entire set of examples from this article in either [Visual Basic](#), [F#](#), or [C#](#).

ⓘ Note

An alternative to the `DateTime` structure for working with date and time values in particular time zones is the `DateTimeOffset` structure. The `DateTimeOffset` structure stores date and time information in a private `DateTime` field and the number of minutes by which that date and time differs from UTC in a private `Int16` field. This makes it possible for a `DateTimeOffset` value to reflect the time in a particular time zone, whereas a `DateTime` value can unambiguously reflect only UTC and the local time zone's time. For a discussion about when to use the `DateTime` structure or the `DateTimeOffset` structure when working with date and time values, see [Choosing Between DateTime, DateTimeOffset, TimeSpan, and TimeZoneInfo](#).

Quick links to example code

ⓘ Note

Some C# examples in this article run in the [Try.NET](#) inline code runner and playground. Select the **Run** button to run an example in an interactive window. Once you execute the code, you can modify it and run the modified code by selecting **Run** again. The modified code either runs in the interactive window or, if compilation fails, the interactive window displays all C# compiler error messages.

The [local time zone](#) of the [Try.NET](#) inline code runner and playground is Coordinated Universal Time, or UTC. This may affect the behavior and the output of examples that illustrate the `DateTime`, `DateTimeOffset`, and `TimeZoneInfo` types and their members.

This article includes several examples that use the `DateTime` type:

Initialization examples

- [Invoke a constructor](#)

- Invoke the implicit parameterless constructor
- Assignment from return value
- Parsing a string that represents a date and time
- Visual Basic syntax to initialize a date and time

Format `DateTime` objects as strings examples

- Use the default date time format
- Format a date and time using a specific culture
- Format a date time using a standard or custom format string
- Specify both a format string and a specific culture
- Format a date time using the ISO 8601 standard for web services

Parse strings as `DateTime` objects examples

- Use Parse or TryParse to convert a string to a date and time
- Use ParseExact or TryParseExact to convert a string in a known format
- Convert from the ISO 8601 string representation to a date and time

`DateTime` resolution examples

- Explore the resolution of date and time values
- Comparing for equality within a tolerance

Culture and calendars examples

- Display date and time values using culture specific calendars
- Parse strings according to a culture specific calendar
- Initialize a date and time from a specific culture's calendar
- Accessing date and time properties using a specific culture's calendar
- Retrieving the week of the year using culture specific calendars

Persistence examples

- Persisting date and time values as strings in the local time zone
- Persisting date and time values as strings in a culture and time invariant format
- Persisting date and time values as integers
- Persisting date and time values using the XmlSerializer

Initialize a DateTime object

You can assign an initial value to a new `DateTime` value in many different ways:

- Calling a constructor, either one where you specify arguments for values, or use the implicit parameterless constructor.
- Assigning a `DateTime` to the return value of a property or method.
- Parsing a `DateTime` value from its string representation.
- Using Visual Basic-specific language features to instantiate a `DateTime`.

The following code snippets show examples of each.

Invoke constructors

You call any of the overloads of the `DateTime` constructor that specify elements of the date and time value (such as the year, month, and day, or the number of ticks). The following code creates a specific date using the `DateTime` constructor specifying the year, month, day, hour, minute, and second.

```
C#
```

```
var date1 = new DateTime(2008, 5, 1, 8, 30, 52);
Console.WriteLine(date1);
```

You invoke the `DateTime` structure's implicit parameterless constructor when you want a `DateTime` initialized to its default value. (For details on the implicit parameterless constructor of a value type, see [Value Types](#).) Some compilers also support declaring a `DateTime` value without explicitly assigning a value to it. Creating a value without an explicit initialization also results in the default value. The following example illustrates the `DateTime` implicit parameterless constructor in C# and Visual Basic, as well as a `DateTime` declaration without assignment in Visual Basic.

```
C#
```

```
var dat1 = new DateTime();
// The following method call displays 1/1/0001 12:00:00 AM.
Console.WriteLine(dat1.ToString(System.Globalization.CultureInfo.InvariantCulture));
// The following method call displays True.
Console.WriteLine(dat1.Equals(DateTime.MinValue));
```

Assign a computed value

You can assign the [DateTime](#) object a date and time value returned by a property or method. The following example assigns the current date and time, the current Coordinated Universal Time (UTC) date and time, and the current date to three new [DateTime](#) variables.

C#

```
DateTime date1 = DateTime.Now;
DateTime date2 = DateTime.UtcNow;
DateTime date3 = DateTime.Today;
```

Parse a string that represents a [DateTime](#)

The [Parse](#), [ParseExact](#), [TryParse](#), and [TryParseExact](#) methods all convert a string to its equivalent date and time value. The following examples use the [Parse](#) and [ParseExact](#) methods to parse a string and convert it to a [DateTime](#) value. The second format uses a form supported by the [ISO 8601](#) standard for a representing date and time in string format. This standard representation is often used to transfer date information in web services.

C#

```
var dateString = "5/1/2008 8:30:52 AM";
DateTime date1 = DateTime.Parse(dateString,
    System.Globalization.CultureInfo.InvariantCulture);

var iso8601String = "20080501T08:30:52Z";
DateTime dateISO8602 = DateTime.ParseExact(iso8601String,
    "yyyyMMddTHH:mm:ssZ",
    System.Globalization.CultureInfo.InvariantCulture);
```

The [TryParse](#) and [TryParseExact](#) methods indicate whether a string is a valid representation of a [DateTime](#) value and, if it is, performs the conversion.

Language-specific syntax for Visual Basic

The following Visual Basic statement initializes a new [DateTime](#) value.

VB

```
Dim date1 As Date = #5/1/2008 8:30:52AM#
```

DateTime values and their string representations

Internally, all [DateTime](#) values are represented as the number of ticks (the number of 100-nanosecond intervals) that have elapsed since 12:00:00 midnight, January 1, 0001. The actual [DateTime](#) value is independent of the way in which that value appears when displayed. The appearance of a [DateTime](#) value is the result of a formatting operation that converts a value to its string representation.

The appearance of date and time values is dependent on culture, international standards, application requirements, and personal preference. The [DateTime](#) structure offers flexibility in formatting date and time values through overloads of [ToString](#). The default [DateTime.ToString\(\)](#) method returns the string representation of a date and time value using the current culture's short date and long time pattern. The following example uses the default [DateTime.ToString\(\)](#) method. It displays the date and time using the short date and long time pattern for the current culture. The en-US culture is the current culture on the computer on which the example was run.

```
C#
```

```
var date1 = new DateTime(2008, 3, 1, 7, 0, 0);
Console.WriteLine(date1.ToString());
// For en-US culture, displays 3/1/2008 7:00:00 AM
```

You may need to format dates in a specific culture to support web scenarios where the server may be in a different culture from the client. You specify the culture using the [DateTime.ToString\(IFormatProvider\)](#) method to create the short date and long time representation in a specific culture. The following example uses the [DateTime.ToString\(IFormatProvider\)](#) method to display the date and time using the short date and long time pattern for the fr-FR culture.

```
C#
```

```
var date1 = new DateTime(2008, 3, 1, 7, 0, 0);
Console.WriteLine(date1.ToString(System.Globalization.CultureInfo.CreateSpecificCulture("fr-FR")));
// Displays 01/03/2008 07:00:00
```

Other applications may require different string representations of a date. The [DateTime.ToString\(String\)](#) method returns the string representation defined by a standard or custom format specifier using the formatting conventions of the current culture. The following example uses the [DateTime.ToString\(String\)](#) method to display the

full date and time pattern for the en-US culture, the current culture on the computer on which the example was run.

C#

```
var date1 = new DateTime(2008, 3, 1, 7, 0, 0);
Console.WriteLine(date1.ToString("F"));
// Displays Saturday, March 01, 2008 7:00:00 AM
```

Finally, you can specify both the culture and the format using the [DateTime.ToString\(String, IFormatProvider\)](#) method. The following example uses the [DateTime.ToString\(String, IFormatProvider\)](#) method to display the full date and time pattern for the fr-FR culture.

C#

```
var date1 = new DateTime(2008, 3, 1, 7, 0, 0);
Console.WriteLine(date1.ToString("F", new
System.Globalization.CultureInfo("fr-FR")));
// Displays samedi 1 mars 2008 07:00:00
```

The [DateTime.ToString\(String\)](#) overload can also be used with a custom format string to specify other formats. The following example shows how to format a string using the [ISO 8601 ↗](#) standard format often used for web services. The Iso 8601 format does not have a corresponding standard format string.

C#

```
var date1 = new DateTime(2008, 3, 1, 7, 0, 0, DateTimeKind.Utc);
Console.WriteLine(date1.ToString("yyyy-MM-ddTHH:mm:sszzz",
System.Globalization.CultureInfo.InvariantCulture));
// Displays 2008-03-01T07:00:00+00:00
```

For more information about formatting [DateTime](#) values, see [Standard Date and Time Format Strings](#) and [Custom Date and Time Format Strings](#).

Parse DateTime values from strings

Parsing converts the string representation of a date and time to a [DateTime](#) value. Typically, date and time strings have two different usages in applications:

- A date and time takes a variety of forms and reflects the conventions of either the current culture or a specific culture. For example, an application allows a user whose current culture is en-US to input a date value as "12/15/2013" or

"December 15, 2013". It allows a user whose current culture is en-gb to input a date value as "15/12/2013" or "15 December 2013."

- A date and time is represented in a predefined format. For example, an application serializes a date as "20130103" independently of the culture on which the app is running. An application may require dates be input in the current culture's short date format.

You use the [Parse](#) or [TryParse](#) method to convert a string from one of the common date and time formats used by a culture to a [DateTime](#) value. The following example shows how you can use [TryParse](#) to convert date strings in different culture-specific formats to a [DateTime](#) value. It changes the current culture to English (United Kingdom) and calls the [GetDateTimeFormats\(\)](#) method to generate an array of date and time strings. It then passes each element in the array to the [TryParse](#) method. The output from the example shows the parsing method was able to successfully convert each of the culture-specific date and time strings.

C#

```
System.Threading.Thread.CurrentThread.CurrentCulture =
    System.Globalization.CultureInfo.CreateSpecificCulture("en-GB");

var date1 = new DateTime(2013, 6, 1, 12, 32, 30);
var badFormats = new List<String>();

Console.WriteLine("${"Date String",-37} {"Date",-19}\n");
foreach (var dateString in date1.GetDateTimeFormats())
{
    DateTime parsedDate;
    if (DateTime.TryParse(dateString, out parsedDate))
        Console.WriteLine($"{dateString,-37}
{DateTime.Parse(dateString),-19}");
    else
        badFormats.Add(dateString);
}

// Display strings that could not be parsed.
if (badFormats.Count > 0)
{
    Console.WriteLine("\nStrings that could not be parsed: ");
    foreach (var badFormat in badFormats)
        Console.WriteLine($"    {badFormat}");
}
// Press "Run" to see the output.
```

You use the [ParseExact](#) and [TryParseExact](#) methods to convert a string that must match a particular format or formats to a [DateTime](#) value. You specify one or more date and time format strings as a parameter to the parsing method. The following example uses the

`TryParseExact(String, String[], IFormatProvider, DateTimeStyles, DateTime)` method to convert strings that must be either in a "yyyyMMdd" format or a "HHmmss" format to `DateTime` values.

C#

```
string[] formats = { "yyyyMMdd", "HHmmss" };
string[] dateStrings = { "20130816", "20131608", " 20130816  ",
                        "115216", "521116", " 115216 " };
DateTime parsedDate;

foreach (var dateString in dateStrings)
{
    if (DateTime.TryParseExact(dateString, formats, null,
        System.Globalization.DateTimeStyles.AllowWhiteSpaces |
        System.Globalization.DateTimeStyles.AdjustToUniversal,
        out parsedDate))
        Console.WriteLine($"{dateString} --> {parsedDate:g}");
    else
        Console.WriteLine($"Cannot convert {dateString}");
}

// The example displays the following output:
//      20130816 --> 8/16/2013 12:00 AM
//      Cannot convert 20131608
//      20130816 --> 8/16/2013 12:00 AM
//      115216 --> 4/22/2013 11:52 AM
//      Cannot convert 521116
//      115216 --> 4/22/2013 11:52 AM
```

One common use for `ParseExact` is to convert a string representation from a web service, usually in [ISO 8601](#) standard format. The following code shows the correct format string to use:

C#

```
var iso8601String = "20080501T08:30:52Z";
DateTime dateISO8602 = DateTime.ParseExact(iso8601String,
    "yyyyMMddTHH:mm:ssZ",
    System.Globalization.CultureInfo.InvariantCulture);
Console.WriteLine($"{iso8601String} --> {dateISO8602:g}");
```

If a string cannot be parsed, the `Parse` and `ParseExact` methods throw an exception. The `TryParse` and `TryParseExact` methods return a `Boolean` value that indicates whether the conversion succeeded or failed. You should use the `TryParse` or `TryParseExact` methods in scenarios where performance is important. The parsing operation for date and time

strings tends to have a high failure rate, and exception handling is expensive. Use these methods if strings are input by users or coming from an unknown source.

For more information about parsing date and time values, see [Parsing Date and Time Strings](#).

Date**Time** values

Descriptions of time values in the [DateTime](#) type are often expressed using the Coordinated Universal Time (UTC) standard. Coordinated Universal Time is the internationally recognized name for Greenwich Mean Time (GMT). Coordinated Universal Time is the time as measured at zero degrees longitude, the UTC origin point. Daylight saving time is not applicable to UTC.

Local time is relative to a particular time zone. A time zone is associated with a time zone offset. A time zone offset is the displacement of the time zone measured in hours from the UTC origin point. In addition, local time is optionally affected by daylight saving time, which adds or subtracts a time interval adjustment. Local time is calculated by adding the time zone offset to UTC and adjusting for daylight saving time if necessary. The time zone offset at the UTC origin point is zero.

UTC time is suitable for calculations, comparisons, and storing dates and time in files. Local time is appropriate for display in user interfaces of desktop applications. Time zone-aware applications (such as many Web applications) also need to work with a number of other time zones.

If the [Kind](#) property of a [DateTime](#) object is [DateTimeKind.Unspecified](#), it is unspecified whether the time represented is local time, UTC time, or a time in some other time zone.

Date**Time** resolution

Note

As an alternative to performing date and time arithmetic on [DateTime](#) values to measure elapsed time, you can use the [Stopwatch](#) class.

The [Ticks](#) property expresses date and time values in units of one ten-millionth of a second. The [Millisecond](#) property returns the thousandths of a second in a date and time value. Using repeated calls to the [DateTime.Now](#) property to measure elapsed time is dependent on the system clock. The system clock on Windows 7 and Windows 8

systems has a resolution of approximately 15 milliseconds. This resolution affects small time intervals less than 100 milliseconds.

The following example illustrates the dependence of current date and time values on the resolution of the system clock. In the example, an outer loop repeats 20 times, and an inner loop serves to delay the outer loop. If the value of the outer loop counter is 10, a call to the [Thread.Sleep](#) method introduces a five-millisecond delay. The following example shows the number of milliseconds returned by the [DateTime.NowMilliseconds](#) property changes only after the call to [Thread.Sleep](#).

C#

```
string output = "";
for (int ctr = 0; ctr <= 20; ctr++)
{
    output += String.Format("${DateTime.Now.Millisecond}\n");
    // Introduce a delay loop.
    for (int delay = 0; delay <= 1000; delay++)
    { }

    if (ctr == 10)
    {
        output += "Thread.Sleep called...\n";
        System.Threading.Thread.Sleep(5);
    }
}
Console.WriteLine(output);
// Press "Run" to see the output.
```

DateTime operations

A calculation using a [DateTime](#) structure, such as [Add](#) or [Subtract](#), does not modify the value of the structure. Instead, the calculation returns a new [DateTime](#) structure whose value is the result of the calculation.

Conversion operations between time zones (such as between UTC and local time, or between one time zone and another) take daylight saving time into account, but arithmetic and comparison operations do not.

The [DateTime](#) structure itself offers limited support for converting from one time zone to another. You can use the [ToLocalTime](#) method to convert UTC to local time, or you can use the [ToUniversalTime](#) method to convert from local time to UTC. However, a full set of time zone conversion methods is available in the [TimeZoneInfo](#) class. You convert the time in any one of the world's time zones to the time in any other time zone using these methods.

Calculations and comparisons of [DateTime](#) objects are meaningful only if the objects represent times in the same time zone. You can use a [TimeZoneInfo](#) object to represent a [DateTime](#) value's time zone, although the two are loosely coupled. A [DateTime](#) object does not have a property that returns an object that represents that date and time value's time zone. The [Kind](#) property indicates if a [DateTime](#) represents UTC, local time, or is unspecified. In a time zone-aware application, you must rely on some external mechanism to determine the time zone in which a [DateTime](#) object was created. You could use a structure that wraps both the [DateTime](#) value and the [TimeZoneInfo](#) object that represents the [DateTime](#) value's time zone. For details on using UTC in calculations and comparisons with [DateTime](#) values, see [Performing Arithmetic Operations with Dates and Times](#).

Each [DateTime](#) member implicitly uses the Gregorian calendar to perform its operation. Exceptions are methods that implicitly specify a calendar. These include constructors that specify a calendar, and methods with a parameter derived from [IFormatProvider](#), such as [System.Globalization.DateTimeFormatInfo](#).

Operations by members of the [DateTime](#) type take into account details such as leap years and the number of days in a month.

DateTime values and calendars

The .NET Class Library includes a number of calendar classes, all of which are derived from the [Calendar](#) class. They are:

- The [ChineseLunisolarCalendar](#) class.
- The [EastAsianLunisolarCalendar](#) class.
- The [GregorianCalendar](#) class.
- The [HebrewCalendar](#) class.
- The [HijriCalendar](#) class.
- The [JapaneseCalendar](#) class.
- The [JapaneseLunisolarCalendar](#) class.
- The [JulianCalendar](#) class.
- The [KoreanCalendar](#) class.
- The [KoreanLunisolarCalendar](#) class.
- The [PersianCalendar](#) class.
- The [TaiwanCalendar](#) class.
- The [TaiwanLunisolarCalendar](#) class.
- The [ThaiBuddhistCalendar](#) class.
- The [UmAlQuraCalendar](#) class.

Important

Eras in the Japanese calendars are based on the emperor's reign and are therefore expected to change. For example, May 1, 2019 marked the beginning of the Reiwa era in the [JapaneseCalendar](#) and [JapaneseLunisolarCalendar](#). Such a change of era affects all applications that use these calendars. For more information and to determine whether your applications are affected, see [Handling a new era in the Japanese calendar in .NET](#). For information on testing your applications on Windows systems to ensure their readiness for the era change, see [Prepare your application for the Japanese era change](#). For features in .NET that support calendars with multiple eras and for best practices when working with calendars that support multiple eras, see [Working with eras](#).

Each culture uses a default calendar defined by its read-only [CultureInfo.Calendar](#) property. Each culture may support one or more calendars defined by its read-only [CultureInfo.OptionalCalendars](#) property. The calendar currently used by a specific [CultureInfo](#) object is defined by its [DateTimeFormatInfo.Calendar](#) property. It must be one of the calendars found in the [CultureInfo.OptionalCalendars](#) array.

A culture's current calendar is used in all formatting operations for that culture. For example, the default calendar of the Thai Buddhist culture is the Thai Buddhist Era calendar, which is represented by the [ThaiBuddhistCalendar](#) class. When a [CultureInfo](#) object that represents the Thai Buddhist culture is used in a date and time formatting operation, the Thai Buddhist Era calendar is used by default. The Gregorian calendar is used only if the culture's [DateTimeFormatInfo.Calendar](#) property is changed, as the following example shows:

C#

```
var thTH = new System.Globalization.CultureInfo("th-TH");
var value = new DateTime(2016, 5, 28);

Console.WriteLine(value.ToString(thTH));

thTH.DateTimeFormat.Calendar = new System.Globalization.GregorianCalendar();
Console.WriteLine(value.ToString(thTH));
// The example displays the following output:
//      28/5/2559 0:00:00
//      28/5/2016 0:00:00
```

A culture's current calendar is also used in all parsing operations for that culture, as the following example shows.

C#

```
var thTH = new System.Globalization.CultureInfo("th-TH");
var value = DateTime.Parse("28/05/2559", thTH);
Console.WriteLine(value.ToString(thTH));

thTH.DateTimeFormat.Calendar = new System.Globalization.GregorianCalendar();
Console.WriteLine(value.ToString(thTH));
// The example displays the following output:
//      28/5/2559 0:00:00
//      28/5/2016 0:00:00
```

You instantiate a [DateTime](#) value using the date and time elements (number of the year, month, and day) of a specific calendar by calling a [DateTime constructor](#) that includes a `calendar` parameter and passing it a [Calendar](#) object that represents that calendar. The following example uses the date and time elements from the [ThaiBuddhistCalendar](#) calendar.

C#

```
var thTH = new System.Globalization.CultureInfo("th-TH");
var dat = new DateTime(2559, 5, 28, thTH.DateTimeFormat.Calendar);
Console.WriteLine($"Thai Buddhist era date: {dat.ToString("d", thTH)}");
Console.WriteLine($"Gregorian date: {dat:d}");
// The example displays the following output:
//      Thai Buddhist Era Date: 28/5/2559
//      Gregorian Date: 28/05/2016
```

[DateTime](#) constructors that do not include a `calendar` parameter assume that the date and time elements are expressed as units in the Gregorian calendar.

All other [DateTime](#) properties and methods use the Gregorian calendar. For example, the [DateTime.Year](#) property returns the year in the Gregorian calendar, and the [DateTime.IsLeapYear\(Int32\)](#) method assumes that the `year` parameter is a year in the Gregorian calendar. Each [DateTime](#) member that uses the Gregorian calendar has a corresponding member of the [Calendar](#) class that uses a specific calendar. For example, the [Calendar.GetYear](#) method returns the year in a specific calendar, and the [Calendar.IsLeapYear](#) method interprets the `year` parameter as a year number in a specific calendar. The following example uses both the [DateTime](#) and the corresponding members of the [ThaiBuddhistCalendar](#) class.

C#

```
var thTH = new System.Globalization.CultureInfo("th-TH");
var cal = thTH.DateTimeFormat.Calendar;
var dat = new DateTime(2559, 5, 28, cal);
```

```

Console.WriteLine("Using the Thai Buddhist Era calendar:");
Console.WriteLine($"Date: {dat.ToString("d", thTH)}");
Console.WriteLine($"Year: {cal.GetYear(dat)}");
Console.WriteLine($"Leap year: {cal.IsLeapYear(cal.GetYear(dat))}\n");

Console.WriteLine("Using the Gregorian calendar:");
Console.WriteLine($"Date: {dat:d}");
Console.WriteLine($"Year: {dat.Year}");
Console.WriteLine($"Leap year: {DateTime.IsLeapYear(dat.Year)}");
// The example displays the following output:
//      Using the Thai Buddhist Era calendar
//      Date : 28/5/2559
//      Year: 2559
//      Leap year : True
//
//      Using the Gregorian calendar
//      Date : 28/05/2016
//      Year: 2016
//      Leap year : True

```

The [DateTime](#) structure includes a [DayOfWeek](#) property that returns the day of the week in the Gregorian calendar. It does not include a member that allows you to retrieve the week number of the year. To retrieve the week of the year, call the individual calendar's [Calendar.GetWeekOfYear](#) method. The following example provides an illustration.

C#

```

var thTH = new System.Globalization.CultureInfo("th-TH");
var thCalendar = thTH.DateTimeFormat.Calendar;
var dat = new DateTime(1395, 8, 18, thCalendar);
Console.WriteLine("Using the Thai Buddhist Era calendar:");
Console.WriteLine($"Date: {dat.ToString("d", thTH)}");
Console.WriteLine($"Day of Week: {thCalendar.GetDayOfWeek(dat)}");
Console.WriteLine($"Week of year: {thCalendar.GetWeekOfYear(dat,
System.Globalization.CalendarWeekRule.FirstDay, DayOfWeek.Sunday)}\n");

var greg = new System.Globalization.GregorianCalendar();
Console.WriteLine("Using the Gregorian calendar:");
Console.WriteLine($"Date: {dat:d}");
Console.WriteLine($"Day of Week: {dat.DayOfWeek}");
Console.WriteLine($"Week of year: {greg.GetWeekOfYear(dat,
System.Globalization.CalendarWeekRule.FirstDay, DayOfWeek.Sunday)}");
// The example displays the following output:
//      Using the Thai Buddhist Era calendar
//      Date : 18/8/1395
//      Day of Week: Sunday
//      Week of year: 34
//
//      Using the Gregorian calendar
//      Date : 18/08/0852

```

```
//      Day of Week: Sunday  
//      Week of year: 34
```

For more information on dates and calendars, see [Working with Calendars](#).

Persist DateTime values

You can persist [DateTime](#) values in the following ways:

- Convert them to strings and persist the strings.
- Convert them to 64-bit integer values (the value of the [Ticks](#) property) and persist the integers.
- Serialize the [DateTime](#) values.

You must ensure that the routine that restores the [DateTime](#) values doesn't lose data or throw an exception regardless of which technique you choose. [DateTime](#) values should round-trip. That is, the original value and the restored value should be the same. And if the original [DateTime](#) value represents a single instant of time, it should identify the same moment of time when it's restored.

Persist values as strings

To successfully restore [DateTime](#) values that are persisted as strings, follow these rules:

- Make the same assumptions about culture-specific formatting when you restore the string as when you persisted it. To ensure that a string can be restored on a system whose current culture is different from the culture of the system it was saved on, call the [ToString](#) overload to save the string by using the conventions of the invariant culture. Call the [Parse\(String, IFormatProvider, DateTimeStyles\)](#) or [TryParse\(String, IFormatProvider, DateTimeStyles, DateTime\)](#) overload to restore the string by using the conventions of the invariant culture. Never use the [ToString\(\)](#), [Parse\(String\)](#), or [TryParse\(String, DateTime\)](#) overloads, which use the conventions of the current culture.
- If the date represents a single moment of time, ensure that it represents the same moment in time when it's restored, even on a different time zone. Convert the [DateTime](#) value to Coordinated Universal Time (UTC) before saving it or use [DateTimeOffset](#).

The most common error made when persisting [DateTime](#) values as strings is to rely on the formatting conventions of the default or current culture. Problems arise if the current culture is different when saving and restoring the strings. The following example

illustrates these problems. It saves five dates using the formatting conventions of the current culture, which in this case is English (United States). It restores the dates using the formatting conventions of a different culture, which in this case is English (United Kingdom). Because the formatting conventions of the two cultures are different, two of the dates can't be restored, and the remaining three dates are interpreted incorrectly. Also, if the original date and time values represent single moments in time, the restored times are incorrect because time zone information is lost.

C#

```
public static void PersistAsLocalStrings()
{
    SaveLocalDatesAsString();
    RestoreLocalDatesFromString();
}

private static void SaveLocalDatesAsString()
{
    DateTime[] dates = { new DateTime(2014, 6, 14, 6, 32, 0),
                         new DateTime(2014, 7, 10, 23, 49, 0),
                         new DateTime(2015, 1, 10, 1, 16, 0),
                         new DateTime(2014, 12, 20, 21, 45, 0),
                         new DateTime(2014, 6, 2, 15, 14, 0) };
    string? output = null;

    Console.WriteLine($"Current Time Zone:
{TimeZoneInfo.Local.DisplayName}");
    Console.WriteLine($"The dates on an
{Thread.CurrentThread.CurrentCulture.Name} system:");
    for (int ctr = 0; ctr < dates.Length; ctr++)
    {
        Console.WriteLine(dates[ctr].ToString("f"));
        output += dates[ctr].ToString() + (ctr != dates.Length - 1 ? "|" :
        ""));
    }
    var sw = new StreamWriter(filenameTxt);
    sw.Write(output);
    sw.Close();
    Console.WriteLine("Saved dates...");
}

private static void RestoreLocalDatesFromString()
{
    TimeZoneInfo.ClearCachedData();
    Console.WriteLine($"Current Time Zone:
{TimeZoneInfo.Local.DisplayName}");
    Thread.CurrentThread.CurrentCulture =
CultureInfo.CreateSpecificCulture("en-GB");
    StreamReader sr = new StreamReader(filenameTxt);
    string[] inputValues = sr.ReadToEnd().Split(new char[] { '|' },
StringSplitOptions.RemoveEmptyEntries);
}
```

```

sr.Close();
Console.WriteLine("The dates on an {0} system:",
                  Thread.CurrentThread.CurrentCulture.Name);
foreach (var inputValue in inputValues)
{
    DateTime dateValue;
    if (DateTime.TryParse(inputValue, out dateValue))
    {
        Console.WriteLine($"{inputValue}' --> {dateValue:f}");
    }
    else
    {
        Console.WriteLine($"Cannot parse '{inputValue}'");
    }
}
Console.WriteLine("Restored dates...");
}

// When saved on an en-US system, the example displays the following output:
//   Current Time Zone: (UTC-08:00) Pacific Time (US & Canada)
//   The dates on an en-US system:
//   Saturday, June 14, 2014 6:32 AM
//   Thursday, July 10, 2014 11:49 PM
//   Saturday, January 10, 2015 1:16 AM
//   Saturday, December 20, 2014 9:45 PM
//   Monday, June 02, 2014 3:14 PM
//   Saved dates...
//
// When restored on an en-GB system, the example displays the following
output:
//   Current Time Zone: (UTC) Dublin, Edinburgh, Lisbon, London
//   The dates on an en-GB system:
//   Cannot parse //6/14/2014 6:32:00 AM//  

//   //7/10/2014 11:49:00 PM// --> 07 October 2014 23:49  

//   //1/10/2015 1:16:00 AM// --> 01 October 2015 01:16  

//   Cannot parse //12/20/2014 9:45:00 PM//  

//   //6/2/2014 3:14:00 PM// --> 06 February 2014 15:14
//   Restored dates...

```

To round-trip [DateTime](#) values successfully, follow these steps:

1. If the values represent single moments of time, convert them from the local time to UTC by calling the [ToUniversalTime](#) method.
2. Convert the dates to their string representations by calling the [ToString\(String, IFormatProvider\)](#) or [String.Format\(IFormatProvider, String, Object\[\]\)](#) overload. Use the formatting conventions of the invariant culture by specifying [CultureInfo.InvariantCulture](#) as the `provider` argument. Specify that the value should round-trip by using the "O" or "R" standard format string.

To restore the persisted [DateTime](#) values without data loss, follow these steps:

1. Parse the data by calling the [ParseExact](#) or [TryParseExact](#) overload. Specify `CultureInfo.InvariantCulture` as the `provider` argument, and use the same standard format string you used for the `format` argument during conversion. Include the `DateTimeStyles.RoundtripKind` value in the `styles` argument.
2. If the `DateTime` values represent single moments in time, call the [ToLocalTime](#) method to convert the parsed date from UTC to local time.

The following example uses the invariant culture and the "O" standard format string to ensure that `DateTime` values saved and restored represent the same moment in time regardless of the system, culture, or time zone of the source and target systems.

C#

```
public static void PersistAsInvariantStrings()
{
    SaveDatesAsInvariantStrings();
    RestoreDatesAsInvariantStrings();
}

private static void SaveDatesAsInvariantStrings()
{
    DateTime[] dates = { new DateTime(2014, 6, 14, 6, 32, 0),
                         new DateTime(2014, 7, 10, 23, 49, 0),
                         new DateTime(2015, 1, 10, 1, 16, 0),
                         new DateTime(2014, 12, 20, 21, 45, 0),
                         new DateTime(2014, 6, 2, 15, 14, 0) };
    string? output = null;

    Console.WriteLine($"Current Time Zone:
{TimeZoneInfo.Local.DisplayName}");
    Console.WriteLine($"The dates on an
{Thread.CurrentThread.CurrentCulture.Name} system:");
    for (int ctr = 0; ctr < dates.Length; ctr++)
    {
        Console.WriteLine(dates[ctr].ToString("f"));
        output += dates[ctr].ToUniversalTime().ToString("O",
CultureInfo.InvariantCulture)
            + (ctr != dates.Length - 1 ? " | " : ""));
    }
    var sw = new StreamWriter(filenameTxt);
    sw.Write(output);
    sw.Close();
    Console.WriteLine("Saved dates...");
}

private static void RestoreDatesAsInvariantStrings()
{
    TimeZoneInfo.ClearCachedData();
    Console.WriteLine("Current Time Zone: {0}",
                      TimeZoneInfo.Local.DisplayName);
    Thread.CurrentThread.CurrentCulture =
}
```

```

CultureInfo.CreateSpecificCulture("en-GB");
StreamReader sr = new StreamReader(filenameTxt);
string[] inputValues = sr.ReadToEnd().Split(new char[] { '|' });

StringSplitOptions.RemoveEmptyEntries);
sr.Close();
Console.WriteLine("The dates on an {0} system:",
                  Thread.CurrentThread.CurrentCulture.Name);
foreach (var inputValue in inputValues)
{
    DateTime dateValue;
    if (DateTime.TryParseExact(inputValue, "0",
CultureInfo.InvariantCulture,
                    DateTimeStyles.RoundtripKind, out dateValue))
    {
        Console.WriteLine($"'{inputValue}' -->
{dateValue.ToLocalTime():f}");
    }
    else
    {
        Console.WriteLine("Cannot parse '{0}'", inputValue);
    }
}
Console.WriteLine("Restored dates...");
}

// When saved on an en-US system, the example displays the following output:
//     Current Time Zone: (UTC-08:00) Pacific Time (US & Canada)
//     The dates on an en-US system:
//     Saturday, June 14, 2014 6:32 AM
//     Thursday, July 10, 2014 11:49 PM
//     Saturday, January 10, 2015 1:16 AM
//     Saturday, December 20, 2014 9:45 PM
//     Monday, June 02, 2014 3:14 PM
//     Saved dates...
//
// When restored on an en-GB system, the example displays the following
output:
//     Current Time Zone: (UTC) Dublin, Edinburgh, Lisbon, London
//     The dates on an en-GB system:
//     '2014-06-14T13:32:00.0000000Z' --> 14 June 2014 14:32
//     '2014-07-11T06:49:00.0000000Z' --> 11 July 2014 07:49
//     '2015-01-10T09:16:00.0000000Z' --> 10 January 2015 09:16
//     '2014-12-21T05:45:00.0000000Z' --> 21 December 2014 05:45
//     '2014-06-02T22:14:00.0000000Z' --> 02 June 2014 23:14
//     Restored dates...

```

Persist values as integers

You can persist a date and time as an [Int64](#) value that represents a number of ticks. In this case, you don't have to consider the culture of the systems the [DateTime](#) values are persisted and restored on.

To persist a [DateTime](#) value as an integer:

1. If the [DateTime](#) values represent single moments in time, convert them to UTC by calling the [ToUniversalTime](#) method.
2. Retrieve the number of ticks represented by the [DateTime](#) value from its [Ticks](#) property.

To restore a [DateTime](#) value that has been persisted as an integer:

1. Instantiate a new [DateTime](#) object by passing the [Int64](#) value to the [DateTime\(Int64\)](#) constructor.
2. If the [DateTime](#) value represents a single moment in time, convert it from UTC to the local time by calling the [ToLocalTime](#) method.

The following example persists an array of [DateTime](#) values as integers on a system in the U.S. Pacific Time zone. It restores it on a system in the UTC zone. The file that contains the integers includes an [Int32](#) value that indicates the total number of [Int64](#) values that immediately follow it.

C#

```
public static void PersistAsIntegers()
{
    SaveDatesAsInts();
    RestoreDatesAsInts();
}

private static void SaveDatesAsInts()
{
    DateTime[] dates = { new DateTime(2014, 6, 14, 6, 32, 0),
                        new DateTime(2014, 7, 10, 23, 49, 0),
                        new DateTime(2015, 1, 10, 1, 16, 0),
                        new DateTime(2014, 12, 20, 21, 45, 0),
                        new DateTime(2014, 6, 2, 15, 14, 0) };

    Console.WriteLine($"Current Time Zone:
{TimeZoneInfo.Local.DisplayName}");
    Console.WriteLine($"The dates on an
{Thread.CurrentThread.CurrentCulture.Name} system:");
    var ticks = new long[dates.Length];
    for (int ctr = 0; ctr < dates.Length; ctr++)
    {
        Console.WriteLine(dates[ctr].ToString("f"));
        ticks[ctr] = dates[ctr].ToUniversalTime().Ticks;
    }
    var fs = new FileStream(filenameInts, FileMode.Create);
    var bw = new BinaryWriter(fs);
    bw.Write(ticks.Length);
    foreach (var tick in ticks)
        bw.Write(tick);
```

```
        bw.Close();
        Console.WriteLine("Saved dates...");
    }

private static void RestoreDatesAsInts()
{
    TimeZoneInfo.ClearCachedData();
    Console.WriteLine($"Current Time Zone:
{TimeZoneInfo.Local.DisplayName}");
    Thread.CurrentThread.CurrentCulture =
CultureInfo.CreateSpecificCulture("en-GB");
    FileStream fs = new FileStream(filenameInts, FileMode.Open);
    BinaryReader br = new BinaryReader(fs);
    int items;
    DateTime[] dates;

    try
    {
        items = br.ReadInt32();
        dates = new DateTime[items];

        for (int ctr = 0; ctr < items; ctr++)
        {
            long ticks = br.ReadInt64();
            dates[ctr] = new DateTime(ticks).ToLocalTime();
        }
    }
    catch (EndOfStreamException)
    {
        Console.WriteLine("File corruption detected. Unable to restore
data...");
        return;
    }
    catch (IOException)
    {
        Console.WriteLine("Unspecified I/O error. Unable to restore
data...");
        return;
    }
    // Thrown during array initialization.
    catch (OutOfMemoryException)
    {
        Console.WriteLine("File corruption detected. Unable to restore
data...");
        return;
    }
    finally
    {
        br.Close();
    }

    Console.WriteLine($"The dates on an
{Thread.CurrentThread.CurrentCulture.Name} system:");
    foreach (var value in dates)
```

```

        Console.WriteLine(value.ToString("f"));

        Console.WriteLine("Restored dates...");
    }
// When saved on an en-US system, the example displays the following output:
//     Current Time Zone: (UTC-08:00) Pacific Time (US & Canada)
//     The dates on an en-US system:
//     Saturday, June 14, 2014 6:32 AM
//     Thursday, July 10, 2014 11:49 PM
//     Saturday, January 10, 2015 1:16 AM
//     Saturday, December 20, 2014 9:45 PM
//     Monday, June 02, 2014 3:14 PM
//     Saved dates...
//
// When restored on an en-GB system, the example displays the following
output:
//     Current Time Zone: (UTC) Dublin, Edinburgh, Lisbon, London
//     The dates on an en-GB system:
//     14 June 2014 14:32
//     11 July 2014 07:49
//     10 January 2015 09:16
//     21 December 2014 05:45
//     02 June 2014 23:14
//     Restored dates...

```

Serialize DateTime values

You can persist [DateTime](#) values through serialization to a stream or file, and then restore them through deserialization. [DateTime](#) data is serialized in some specified object format. The objects are restored when they are deserialized. A formatter or serializer, such as [JsonSerializer](#) or [XmlSerializer](#), handles the process of serialization and deserialization. For more information about serialization and the types of serialization supported by .NET, see [Serialization](#).

The following example uses the [XmlSerializer](#) class to serialize and deserialize [DateTime](#) values. The values represent all leap year days in the twenty-first century. The output represents the result if the example is run on a system whose current culture is English (United Kingdom). Because you've deserialized the [DateTime](#) object itself, the code doesn't have to handle cultural differences in date and time formats.

C#

```

public static void PersistAsXML()
{
    // Serialize the data.
    var leapYears = new List<DateTime>();
    for (int year = 2000; year <= 2100; year += 4)
    {
        if (DateTime.IsLeapYear(year))

```

```

        leapYears.Add(new DateTime(year, 2, 29));
    }
    DateTime[] dateArray = leapYears.ToArray();

    var serializer = new XmlSerializer(dateArray.GetType());
    TextWriter sw = new StreamWriter(filenameXml);

    try
    {
        serializer.Serialize(sw, dateArray);
    }
    catch (InvalidOperationException e)
    {
        Console.WriteLine(e.InnerException?.Message);
    }
    finally
    {
        if (sw != null) sw.Close();
    }

    // Deserialize the data.
    DateTime[]? deserializedDates;
    using (var fs = new FileStream(filenameXml, FileMode.Open))
    {
        deserializedDates = (DateTime[]?)serializer.Deserialize(fs);
    }

    // Display the dates.
    Console.WriteLine($"Leap year days from 2000-2100 on an
{Thread.CurrentThread.CurrentCulture.Name} system:");
    int nItems = 0;
    if (deserializedDates is not null)
    {
        foreach (var dat in deserializedDates)
        {
            Console.Write($" {dat:d} ");
            nItems++;
            if (nItems % 5 == 0)
                Console.WriteLine();
        }
    }
}

// The example displays the following output:
// Leap year days from 2000-2100 on an en-GB system:
// 29/02/2000 29/02/2004 29/02/2008 29/02/2012
29/02/2016
// 29/02/2020 29/02/2024 29/02/2028 29/02/2032
29/02/2036
// 29/02/2040 29/02/2044 29/02/2048 29/02/2052
29/02/2056
// 29/02/2060 29/02/2064 29/02/2068 29/02/2072
29/02/2076
// 29/02/2080 29/02/2084 29/02/2088 29/02/2092
29/02/2096

```

The previous example doesn't include time information. If a `DateTime` value represents a moment in time and is expressed as a local time, convert it from local time to UTC before serializing it by calling the `ToUniversalTime` method. After you deserialize it, convert it from UTC to local time by calling the `ToLocalTime` method.

DateTime vs. TimeSpan

The `DateTime` and `TimeSpan` value types differ in that a `DateTime` represents an instant in time whereas a `TimeSpan` represents a time interval. You can subtract one instance of `DateTime` from another to obtain a `TimeSpan` object that represents the time interval between them. Or you could add a positive `TimeSpan` to the current `DateTime` to obtain a `DateTime` value that represents a future date.

You can add or subtract a time interval from a `DateTime` object. Time intervals can be negative or positive, and they can be expressed in units such as ticks, seconds, or as a `TimeSpan` object.

Compare for equality within tolerance

Equality comparisons for `DateTime` values are exact. To be considered equal, two values must be expressed as the same number of ticks. That precision is often unnecessary or even incorrect for many applications. Often, you want to test if `DateTime` objects are **roughly equal**.

The following example demonstrates how to compare roughly equivalent `DateTime` values. It accepts a small margin of difference when declaring them equal.

C#

```
public static bool RoughlyEquals(DateTime time, DateTime timeWithWindow, int windowInSeconds, int frequencyInSeconds)
{
    long delta = (long)((TimeSpan)(timeWithWindow - time)).TotalSeconds % frequencyInSeconds;
    delta = delta > windowInSeconds ? frequencyInSeconds - delta : delta;
    return Math.Abs(delta) < windowInSeconds;
}

public static void TestRoughlyEquals()
{
    int window = 10;
    int freq = 60 * 60 * 2; // 2 hours;

    DateTime d1 = DateTime.Now;
```

```

DateTime d2 = d1.AddSeconds(2 * window);
DateTime d3 = d1.AddSeconds(-2 * window);
DateTime d4 = d1.AddSeconds(window / 2);
DateTime d5 = d1.AddSeconds(-window / 2);

DateTime d6 = (d1.AddHours(2)).AddSeconds(2 * window);
DateTime d7 = (d1.AddHours(2)).AddSeconds(-2 * window);
DateTime d8 = (d1.AddHours(2)).AddSeconds(window / 2);
DateTime d9 = (d1.AddHours(2)).AddSeconds(-window / 2);

Console.WriteLine($"d1 ({d1}) ~= d1 ({d1}): {RoughlyEquals(d1, d1,
window, freq)}");
Console.WriteLine($"d1 ({d1}) ~= d2 ({d2}): {RoughlyEquals(d1, d2,
window, freq)}");
Console.WriteLine($"d1 ({d1}) ~= d3 ({d3}): {RoughlyEquals(d1, d3,
window, freq)}");
Console.WriteLine($"d1 ({d1}) ~= d4 ({d4}): {RoughlyEquals(d1, d4,
window, freq)}");
Console.WriteLine($"d1 ({d1}) ~= d5 ({d5}): {RoughlyEquals(d1, d5,
window, freq)}");

Console.WriteLine($"d1 ({d1}) ~= d6 ({d6}): {RoughlyEquals(d1, d6,
window, freq)}");
Console.WriteLine($"d1 ({d1}) ~= d7 ({d7}): {RoughlyEquals(d1, d7,
window, freq)}");
Console.WriteLine($"d1 ({d1}) ~= d8 ({d8}): {RoughlyEquals(d1, d8,
window, freq)}");
Console.WriteLine($"d1 ({d1}) ~= d9 ({d9}): {RoughlyEquals(d1, d9,
window, freq)}");
}

// The example displays output similar to the following:
//      d1 (1/28/2010 9:01:26 PM) ~= d1 (1/28/2010 9:01:26 PM): True
//      d1 (1/28/2010 9:01:26 PM) ~= d2 (1/28/2010 9:01:46 PM): False
//      d1 (1/28/2010 9:01:26 PM) ~= d3 (1/28/2010 9:01:06 PM): False
//      d1 (1/28/2010 9:01:26 PM) ~= d4 (1/28/2010 9:01:31 PM): True
//      d1 (1/28/2010 9:01:26 PM) ~= d5 (1/28/2010 9:01:21 PM): True
//      d1 (1/28/2010 9:01:26 PM) ~= d6 (1/28/2010 11:01:46 PM): False
//      d1 (1/28/2010 9:01:26 PM) ~= d7 (1/28/2010 11:01:06 PM): False
//      d1 (1/28/2010 9:01:26 PM) ~= d8 (1/28/2010 11:01:31 PM): True
//      d1 (1/28/2010 9:01:26 PM) ~= d9 (1/28/2010 11:01:21 PM): True

```

COM interop considerations

A [DateTime](#) value that is transferred to a COM application, then is transferred back to a managed application, is said to round-trip. However, a [DateTime](#) value that specifies only a time does not round-trip as you might expect.

If you round-trip only a time, such as 3 P.M., the final date and time is December 30, 1899 C.E. at 3:00 P.M., instead of January, 1, 0001 C.E. at 3:00 P.M. .NET and COM assume a default date when only a time is specified. However, the COM system assumes

a base date of December 30, 1899 C.E., while .NET assumes a base date of January, 1, 0001 C.E.

When only a time is passed from .NET to COM, special processing is performed that converts the time to the format used by COM. When only a time is passed from COM to .NET, no special processing is performed because that would corrupt legitimate dates and times on or before December 30, 1899. If a date starts its round-trip from COM, .NET and COM preserve the date.

The behavior of .NET and COM means that if your application round-trips a [DateTime](#) that only specifies a time, your application must remember to modify or ignore the erroneous date from the final [DateTime](#) object.

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

.NET is an open source project.
Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

System.TimeSpan struct

Article • 01/08/2024

This article provides supplementary remarks to the reference documentation for this API.

A [TimeSpan](#) object represents a time interval (duration of time or elapsed time) that is measured as a positive or negative number of days, hours, minutes, seconds, and fractions of a second. The [TimeSpan](#) structure can also be used to represent the time of day, but only if the time is unrelated to a particular date. Otherwise, the [DateTime](#) or [DateTimeOffset](#) structure should be used instead. (For more information about using the [TimeSpan](#) structure to reflect the time of day, see [Choosing Between DateTime, DateTimeOffset, TimeSpan, and TimeZoneInfo](#).)

ⓘ Note

A [TimeSpan](#) value represents a time interval and can be expressed as a particular number of days, hours, minutes, seconds, and milliseconds. Because it represents a general interval without reference to a particular start or end point, it cannot be expressed in terms of years and months, both of which have a variable number of days. It differs from a [DateTime](#) value, which represents a date and time without reference to a particular time zone, or a [DateTimeOffset](#) value, which represents a specific moment of time.

The largest unit of time that the [TimeSpan](#) structure uses to measure duration is a day. Time intervals are measured in days for consistency, because the number of days in larger units of time, such as months and years, varies.

The value of a [TimeSpan](#) object is the number of ticks that equal the represented time interval. A tick is equal to 100 nanoseconds, or one ten-millionth of a second. The value of a [TimeSpan](#) object can range from [TimeSpan.MinValue](#) to [TimeSpan.MaxValue](#).

Instantiate a TimeSpan value

You can instantiate a [TimeSpan](#) value in a number of ways:

- By calling its implicit parameterless constructor. This creates an object whose value is [TimeSpan.Zero](#), as the following example shows.

C#

```
TimeSpan interval = new TimeSpan();
Console.WriteLine(interval.Equals(TimeSpan.Zero));      // Displays
"True".
```

- By calling one of its explicit constructors. The following example initializes a [TimeSpan](#) value to a specified number of hours, minutes, and seconds.

C#

```
TimeSpan interval = new TimeSpan(2, 14, 18);
Console.WriteLine(interval.ToString());
// Displays "02:14:18".
```

- By calling a method or performing an operation that returns a [TimeSpan](#) value. For example, you can instantiate a [TimeSpan](#) value that represents the interval between two date and time values, as the following example shows.

C#

```
DateTime departure = new DateTime(2010, 6, 12, 18, 32, 0);
DateTime arrival = new DateTime(2010, 6, 13, 22, 47, 0);
TimeSpan travelTime = arrival - departure;
Console.WriteLine("{0} - {1} = {2}", arrival, departure, travelTime);

// The example displays the following output:
//       6/13/2010 10:47:00 PM - 6/12/2010 6:32:00 PM = 1.04:15:00
```

You can also initialize a [TimeSpan](#) object to a zero time value in this way, as the following example shows.

C#

```
Random rnd = new Random();

TimeSpan timeSpent = TimeSpan.Zero;

timeSpent += GetTimeBeforeLunch();
timeSpent += GetTimeAfterLunch();

Console.WriteLine("Total time: {0}", timeSpent);

TimeSpan GetTimeBeforeLunch()
{
    return new TimeSpan(rnd.Next(3, 6), 0, 0);
}

TimeSpan GetTimeAfterLunch()
```

```

{
    return new TimeSpan(rnd.Next(3, 6), 0, 0);
}

// The example displays output like the following:
//      Total time: 08:00:00

```

`TimeSpan` values are returned by arithmetic operators and methods of the `DateTime`, `DateTimeOffset`, and `TimeSpan` structures.

- By parsing the string representation of a `TimeSpan` value. You can use the `Parse` and `TryParse` methods to convert strings that contain time intervals to `TimeSpan` values. The following example uses the `Parse` method to convert an array of strings to `TimeSpan` values.

C#

```

string[] values = { "12", "31.", "5.8:32:16", "12:12:15.95", ".12"};
foreach (string value in values)
{
    try {
        TimeSpan ts = TimeSpan.Parse(value);
        Console.WriteLine("{0} --> {1}", value, ts);
    }
    catch (FormatException) {
        Console.WriteLine("Unable to parse '{0}'", value);
    }
    catch (OverflowException) {
        Console.WriteLine("{0} is outside the range of a TimeSpan.", value);
    }
}

// The example displays the following output:
//      '12' --> 12.00:00:00
//      Unable to parse '31.'
//      '5.8:32:16' --> 5.08:32:16
//      '12:12:15.95' --> 12:12:15.9500000
//      Unable to parse '.12'

```

In addition, you can define the precise format of the input string to be parsed and converted to a `TimeSpan` value by calling the `ParseExact` or `TryParseExact` method.

Perform operations on `TimeSpan` values

You can add and subtract time durations either by using the `Addition` and `Subtraction` operators, or by calling the `Add` and `Subtract` methods. You can also compare two time durations by calling the `Compare`, `CompareTo`, and `Equals` methods. The `TimeSpan`

structure also includes the [Duration](#) and [Negate](#) methods, which convert time intervals to positive and negative values,

The range of [TimeSpan](#) values is [MinValue](#) to [MaxValue](#).

Format a TimeSpan value

A [TimeSpan](#) value can be represented as $[-]d.hh:mm:ss.ff$, where the optional minus sign indicates a negative time interval, the d component is days, hh is hours as measured on a 24-hour clock, mm is minutes, ss is seconds, and ff is fractions of a second. That is, a time interval consists of a positive or negative number of days without a time of day, or a number of days with a time of day, or only a time of day.

Beginning with .NET Framework 4, the [TimeSpan](#) structure supports culture-sensitive formatting through the overloads of its [ToString](#) method, which converts a [TimeSpan](#) value to its string representation. The default [TimeSpan.ToString\(\)](#) method returns a time interval by using an invariant format that is identical to its return value in previous versions of .NET Framework. The [TimeSpan.ToString\(String\)](#) overload lets you specify a format string that defines the string representation of the time interval. The [TimeSpan.ToString\(String, IFormatProvider\)](#) overload lets you specify a format string and the culture whose formatting conventions are used to create the string representation of the time interval. [TimeSpan](#) supports both standard and custom format strings. (For more information, see [Standard TimeSpan Format Strings](#) and [Custom TimeSpan Format Strings](#).) However, only standard format strings are culture-sensitive.

Restore legacy TimeSpan formatting

In some cases, code that successfully formats [TimeSpan](#) values in .NET Framework 3.5 and earlier versions fails in .NET Framework 4. This is most common in code that calls a [`<TimeSpan_LegacyFormatMode>`](#) element method to format a [TimeSpan](#) value with a format string. The following example successfully formats a [TimeSpan](#) value in .NET Framework 3.5 and earlier versions, but throws an exception in .NET Framework 4 and later versions. Note that it attempts to format a [TimeSpan](#) value by using an unsupported format specifier, which is ignored in .NET Framework 3.5 and earlier versions.

C#

```
ShowFormattingCode();
// Output from .NET Framework 3.5 and earlier versions:
//      12:30:45
// Output from .NET Framework 4:
```

```

//      Invalid Format

Console.WriteLine("---");

ShowParsingCode();
// Output:
//      00000006 --> 6.00:00:00

void ShowFormattingCode()
{
    TimeSpan interval = new TimeSpan(12, 30, 45);
    string output;
    try
    {
        output = String.Format("{0:r}", interval);
    }
    catch (FormatException)
    {
        output = "Invalid Format";
    }
    Console.WriteLine(output);
}

void ShowParsingCode()
{
    string value = "000000006";
    try
    {
        TimeSpan interval = TimeSpan.Parse(value);
        Console.WriteLine("{0} --> {1}", value, interval);
    }
    catch (FormatException)
    {
        Console.WriteLine("{0}: Bad Format", value);
    }
    catch (OverflowException)
    {
        Console.WriteLine("{0}: Overflow", value);
    }
}

```

If you cannot modify the code, you can restore the legacy formatting of [TimeSpan](#) values in one of the following ways:

- By creating a configuration file that contains the [`<TimeSpan_LegacyFormatMode>`](#) element. Setting this element's `enabled` attribute to `true` restores legacy [TimeSpan](#) formatting on a per-application basis.
- By setting the "NetFx40_TimeSpanLegacyFormatMode" compatibility switch when you create an application domain. This enables legacy [TimeSpan](#) formatting on a

per-application-domain basis. The following example creates an application domain that uses legacy [TimeSpan](#) formatting.

```
C#  
  
using System;  
  
public class Example2  
{  
    public static void Main()  
    {  
        AppDomainSetup appSetup = new AppDomainSetup();  
        appSetup.SetCompatibilitySwitches(new string[] {  
            "NetFx40_TimeSpanLegacyFormatMode" });  
        AppDomain legacyDomain = AppDomain.CreateDomain("legacyDomain",  
                                                       null,  
                                                       appSetup);  
        legacyDomain.ExecuteAssembly("ShowTimeSpan.exe");  
    }  
}
```

When the following code executes in the new application domain, it reverts to legacy [TimeSpan](#) formatting behavior.

```
C#  
  
using System;  
  
public class Example3  
{  
    public static void Main()  
    {  
        TimeSpan interval = DateTime.Now - DateTime.Now.Date;  
        string msg = String.Format("Elapsed Time Today: {0:d} hours.",  
                                   interval);  
        Console.WriteLine(msg);  
    }  
}  
// The example displays the following output:  
//     Elapsed Time Today: 01:40:52.2524662 hours.
```

 Collaborate with us on
GitHub

The source for this content can
be found on GitHub, where you
can also create and review

.NET

[.NET feedback](#)

.NET is an open source project.
Select a link to provide feedback:

 [Open a documentation issue](#)

issues and pull requests. For more information, see [our contributor guide](#).

 [Provide product feedback](#)

Extend metadata using attributes

Article • 09/15/2021

The common language runtime allows you to add keyword-like descriptive declarations, called attributes, to annotate programming elements such as types, fields, methods, and properties. When you compile your code for the runtime, it is converted into Microsoft intermediate language (MSIL) and placed inside a portable executable (PE) file along with metadata generated by the compiler. Attributes allow you to place extra descriptive information into metadata that can be extracted using runtime reflection services. The compiler creates attributes when you declare instances of special classes that derive from [System.Attribute](#).

.NET uses attributes for a variety of reasons and to address a number of issues. Attributes describe how to serialize data, specify characteristics that are used to enforce security, and limit optimizations by the just-in-time (JIT) compiler so the code remains easy to debug. Attributes can also record the name of a file or the author of code, or control the visibility of controls and members during forms development.

Related articles

| Title | Description |
|--|---|
| Applying Attributes | Describes how to apply an attribute to an element of your code. |
| Writing Custom Attributes | Describes how to design custom attribute classes. |
| Retrieving Information Stored in Attributes | Describes how to retrieve custom attributes for code that is loaded into the execution context. |
| Metadata and Self-Describing Components | Provides an overview of metadata and describes how it is implemented in a .NET portable executable (PE) file. |
| How to: Load Assemblies into the Reflection-Only Context | Explains how to retrieve custom attribute information in the reflection-only context. |

Reference

- [System.Attribute](#)

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

Apply attributes

Article • 03/15/2023

Use the following process to apply an attribute to an element of your code.

1. Define a new attribute or use an existing .NET attribute.
2. Apply the attribute to the code element by placing it immediately before the element.

Each language has its own attribute syntax. In C++ and C#, the attribute is surrounded by square brackets and separated from the element by white space, which can include a line break. In Visual Basic, the attribute is surrounded by angle brackets and must be on the same logical line; the line continuation character can be used if a line break is desired.

3. Specify positional parameters and named parameters for the attribute.

Positional parameters are required and must come before any named parameters; they correspond to the parameters of one of the attribute's constructors. *Named* parameters are optional and correspond to read/write properties of the attribute. In C++, and C#, specify `name=value` for each optional parameter, where `name` is the name of the property. In Visual Basic, specify `name:=value`.

The attribute is emitted into metadata when you compile your code and is available to the common language runtime and any custom tool or application through the runtime reflection services.

By convention, all attribute names end with "Attribute". However, several languages that target the runtime, such as Visual Basic and C#, do not require you to specify the full name of an attribute. For example, if you want to initialize [System.ObsoleteAttribute](#), you only need to reference it as **Obsolete**.

Apply an attribute to a method

The following code example shows how to use [System.ObsoleteAttribute](#), which marks code as obsolete. The string `"Will be removed in next version"` is passed to the attribute. This attribute causes a compiler warning that displays the passed string when code that the attribute describes is called.

C#

```
public class Example
{
    // Specify attributes between square brackets in C#.
    // This attribute is applied only to the Add method.
    [Obsolete("Will be removed in next version.")]
    public static int Add(int a, int b)
    {
        return (a + b);
    }
}

class Test
{
    public static void Main()
    {
        // This generates a compile-time warning.
        int i = Example.Add(2, 2);
    }
}
```

Apply attributes at the assembly level

If you want to apply an attribute at the assembly level, use the `assembly` (`Assembly` in Visual Basic) keyword. The following code shows the `AssemblyTitleAttribute` applied at the assembly level.

C#

```
using System.Reflection;
[assembly: AssemblyTitle("My Assembly")]
```

When this attribute is applied, the string `"My Assembly"` is placed in the assembly manifest in the metadata portion of the file. You can view the attribute either by using the [MSIL Disassembler \(Ildasm.exe\)](#) or by creating a custom program to retrieve the attribute.

See also

- [Attributes](#)
- [Retrieving Information Stored in Attributes](#)
- [Concepts](#)
- [Attributes \(C#\)](#)
- [Attributes overview \(Visual Basic\)](#)

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

Write custom attributes

Article • 04/12/2023

To design custom attributes, you don't need to learn many new concepts. If you're familiar with object-oriented programming and know how to design classes, you already have most of the knowledge needed. Custom attributes are traditional classes that derive directly or indirectly from the [System.Attribute](#) class. Just like traditional classes, custom attributes contain methods that store and retrieve data.

The primary steps to properly design custom attribute classes are as follows:

- [Applying the AttributeUsageAttribute](#)
- [Declaring the attribute class](#)
- [Declaring constructors](#)
- [Declaring properties](#)

This section describes each of these steps and concludes with a [custom attribute example](#).

Applying the AttributeUsageAttribute

A custom attribute declaration begins with the [System.AttributeUsageAttribute](#) attribute, which defines some of the key characteristics of your attribute class. For example, you can specify whether your attribute can be inherited by other classes or which elements the attribute can be applied to. The following code fragment demonstrates how to use the [AttributeUsageAttribute](#):

C#

```
[AttributeUsage(AttributeTargets.All, Inherited = false, AllowMultiple = true)]
```

The [AttributeUsageAttribute](#) has three members that are important for the creation of custom attributes: [AttributeTargets](#), [Inherited](#), and [AllowMultiple](#).

AttributeTargets Member

In the preceding example, [AttributeTargets.All](#) is specified, indicating that this attribute can be applied to all program elements. Alternatively, you can specify

`AttributeTargets.Class`, indicating that your attribute can be applied only to a class, or `AttributeTargets.Method`, indicating that your attribute can be applied only to a method. All program elements can be marked for description by a custom attribute in this manner.

You can also pass multiple `AttributeTargets` values. The following code fragment specifies that a custom attribute can be applied to any class or method:

```
C#
```

```
[AttributeUsage(AttributeTargets.Class | AttributeTargets.Method)]
```

Inherited Property

The `AttributeUsageAttribute.Inherited` property indicates whether your attribute can be inherited by classes that are derived from the classes to which your attribute is applied. This property takes either a `true` (the default) or `false` flag. In the following example, `MyAttribute` has a default `Inherited` value of `true`, while `YourAttribute` has an `Inherited` value of `false`:

```
C#
```

```
// This defaults to Inherited = true.
public class MyAttribute : Attribute
{
    //...
}

[AttributeUsage(AttributeTargets.Method, Inherited = false)]
public class YourAttribute : Attribute
{
    //...
}
```

The two attributes are then applied to a method in the base class `MyClass`:

```
C#
```

```
public class MyClass
{
    [MyAttribute]
    [YourAttribute]
    public virtual void MyMethod()
    {
        //...
    }
}
```

```
    }  
}
```

Finally, the class `YourClass` is inherited from the base class `MyClass`. The method `MyMethod` shows `MyAttribute` but not `YourAttribute`:

C#

```
public class YourClass : MyClass  
{  
    // MyMethod will have MyAttribute but not YourAttribute.  
    public override void MyMethod()  
    {  
        //...  
    }  
}
```

AllowMultiple Property

The `AttributeUsageAttribute.AllowMultiple` property indicates whether multiple instances of your attribute can exist on an element. If set to `true`, multiple instances are allowed. If set to `false` (the default), only one instance is allowed.

In the following example, `MyAttribute` has a default `AllowMultiple` value of `false`, while `YourAttribute` has a value of `true`:

C#

```
//This defaults to AllowMultiple = false.  
public class MyAttribute : Attribute  
{  
}  
  
[AttributeUsage(AttributeTargets.Method, AllowMultiple = true)]  
public class YourAttribute : Attribute  
{  
}
```

When multiple instances of these attributes are applied, `MyAttribute` produces a compiler error. The following code example shows the valid use of `YourAttribute` and the invalid use of `MyAttribute`:

C#

```
public class MyClass  
{
```

```

// This produces an error.
// Duplicates are not allowed.
[MyAttribute]
[MyAttribute]
public void MyMethod()
{
    //...
}

// This is valid.
[YourAttribute]
[YourAttribute]
public void YourMethod()
{
    //...
}

```

If both the `AllowMultiple` property and the `Inherited` property are set to `true`, a class that's inherited from another class can inherit an attribute and have another instance of the same attribute applied in the same child class. If `AllowMultiple` is set to `false`, the values of any attributes in the parent class will be overwritten by new instances of the same attribute in the child class.

Declaring the Attribute Class

After you apply the `AttributeUsageAttribute`, start defining the specifics of your attribute. The declaration of an attribute class looks similar to the declaration of a traditional class, as demonstrated by the following code:

C#

```

[AttributeUsage(AttributeTargets.Method)]
public class MyAttribute : Attribute
{
    // . . .
}
```

This attribute definition demonstrates the following points:

- Attribute classes must be declared as public classes.
- By convention, the name of the attribute class ends with the word **Attribute**. While not required, this convention is recommended for readability. When the attribute is applied, the inclusion of the word **Attribute** is optional.

- All attribute classes must inherit directly or indirectly from the [System.Attribute](#) class.
- In Microsoft Visual Basic, all custom attribute classes must have the [System.AttributeUsageAttribute](#) attribute.

Declaring Constructors

Just like traditional classes, attributes are initialized with constructors. The following code fragment illustrates a typical attribute constructor. This public constructor takes a parameter and sets a member variable equal to its value.

C#

```
public MyAttribute(bool myvalue)
{
    this.myvalue = myvalue;
}
```

You can overload the constructor to accommodate different combinations of values. If you also define a [property](#) for your custom attribute class, you can use a combination of named and positional parameters when initializing the attribute. Typically, you define all required parameters as positional and all optional parameters as named. In this case, the attribute can't be initialized without the required parameter. All other parameters are optional.

 **Note**

In Visual Basic, constructors for an attribute class shouldn't use a `ParamArray` argument.

The following code example shows how an attribute that uses the previous constructor can be applied using optional and required parameters. It assumes that the attribute has one required Boolean value and one optional string property.

C#

```
// One required (positional) and one optional (named) parameter are applied.
[MyAttribute(false, OptionalParameter = "optional data")]
public class SomeClass
{
    //...
}
// One required (positional) parameter is applied.
```

```
[MyAttribute(false)]
public class SomeOtherClass
{
    //...
}
```

Declaring Properties

If you want to define a named parameter or provide an easy way to return the values stored by your attribute, declare a [property](#). Attribute properties should be declared as public entities with a description of the data type that will be returned. Define the variable that will hold the value of your property and associate it with the `get` and `set` methods. The following code example demonstrates how to implement a property in your attribute:

C#

```
public bool MyProperty
{
    get {return this.myvalue;}
    set {this.myvalue = value;}
}
```

Custom Attribute Example

This section incorporates the previous information and shows how to design an attribute that documents information about the author of a section of code. The attribute in this example stores the name and level of the programmer, and whether the code has been reviewed. It uses three private variables to store the actual values to save. Each variable is represented by a public property that gets and sets the values. Finally, the constructor is defined with two required parameters:

C#

```
[AttributeUsage(AttributeTargets.All)]
public class DeveloperAttribute : Attribute
{
    // Private fields.
    private string name;
    private string level;
    private bool reviewed;

    // This constructor defines two required parameters: name and level.

    public DeveloperAttribute(string name, string level)
```

```

{
    this.name = name;
    this.level = level;
    this.reviewed = false;
}

// Define Name property.
// This is a read-only attribute.

public virtual string Name
{
    get {return name;}
}

// Define Level property.
// This is a read-only attribute.

public virtual string Level
{
    get {return level;}
}

// Define Reviewed property.
// This is a read/write attribute.

public virtual bool Reviewed
{
    get {return reviewed;}
    set {reviewed = value;}
}
}

```

You can apply this attribute using the full name, `DeveloperAttribute`, or using the abbreviated name, `Developer`, in one of the following ways:

C#

```

[Developer("Joan Smith", "1")]

-or-

[Developer("Joan Smith", "1", Reviewed = true)]

```

The first example shows the attribute applied with only the required named parameters. The second example shows the attribute applied with both the required and optional parameters.

See also

- [System.Attribute](#)
- [System.AttributeUsageAttribute](#)
- [Attributes](#)
- [Attribute parameter types](#)

 **Collaborate with us on GitHub**

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

Retrieving Information Stored in Attributes

Article • 10/04/2022

Retrieving a custom attribute is a simple process. First, declare an instance of the attribute you want to retrieve. Then, use the [Attribute.GetCustomAttribute](#) method to initialize the new attribute to the value of the attribute you want to retrieve. Once the new attribute is initialized, you can use its properties to get the values.

ⓘ Important

This article describes how to retrieve attributes for code loaded into the execution context. To retrieve attributes for code loaded into the reflection-only context, you must use the [CustomAttributeData](#) class, as shown in [How to: Load Assemblies into the Reflection-Only Context](#).

This section describes the following ways to retrieve attributes:

- [Retrieving a single instance of an attribute](#)
- [Retrieving multiple instances of an attribute applied to the same scope](#)
- [Retrieving multiple instances of an attribute applied to different scopes](#)

Retrieving a Single Instance of an Attribute

In the following example, the `DeveloperAttribute` (described in the previous section) is applied to the `MainApp` class on the class level. The `GetAttribute` method uses `GetCustomAttribute` to retrieve the values stored in `DeveloperAttribute` on the class level before displaying them to the console.

C#

```
using System;
using System.Reflection;
using CustomCodeAttributes;

[Developer("Joan Smith", "42", Reviewed = true)]
class MainApp
{
    public static void Main()
    {
```

```

        // Call function to get and display the attribute.
        GetAttribute(typeof(MainApp));
    }

    public static void GetAttribute(Type t)
    {
        // Get instance of the attribute.
        DeveloperAttribute MyAttribute =
            (DeveloperAttribute) Attribute.GetCustomAttribute(t, typeof
(DeveloperAttribute));

        if (MyAttribute == null)
        {
            Console.WriteLine("The attribute was not found.");
        }
        else
        {
            // Get the Name value.
            Console.WriteLine("The Name Attribute is: {0}." ,
MyAttribute.Name);
            // Get the Level value.
            Console.WriteLine("The Level Attribute is: {0}." ,
MyAttribute.Level);
            // Get the Reviewed value.
            Console.WriteLine("The Reviewed Attribute is: {0}." ,
MyAttribute.Reviewed);
        }
    }
}

```

The execution of the preceding program displays the following text:

```

Console

The Name Attribute is: Joan Smith.
The Level Attribute is: 42.
The Reviewed Attribute is: True.

```

If the attribute isn't found, the `GetCustomAttribute` method initializes `MyAttribute` to a null value. This example checks `MyAttribute` for such an instance and notifies the user if the attribute isn't found. If `DeveloperAttribute` isn't found in the class scope, the console displays the following message:

```

Console

The attribute was not found.

```

The preceding example assumes that the attribute definition is in the current namespace. Remember to import the namespace in which the attribute definition

resides if it isn't in the current namespace.

Retrieving Multiple Instances of an Attribute Applied to the Same Scope

In the preceding example, the class to inspect and the specific attribute to find are passed to the `GetCustomAttribute` method. That code works well if only one instance of an attribute is applied on the class level. However, if multiple instances of an attribute are applied on the same class level, the `GetCustomAttribute` method doesn't retrieve all the information. In cases where multiple instances of the same attribute are applied to the same scope, you can use `Attribute.GetCustomAttributes` method to place all instances of an attribute into an array. For example, if two instances of `DeveloperAttribute` are applied on the class level of the same class, the `GetAttribute` method can be modified to display the information found in both attributes. Remember, to apply multiple attributes on the same level, the attribute must be defined with the `AllowMultiple` property set to `true` in the `AttributeUsageAttribute` class.

The following code example shows how to use the `GetCustomAttributes` method to create an array that references all instances of `DeveloperAttribute` in any given class. The code then outputs the values of all the attributes to the console.

C#

```
public static void GetAttribute(Type t)
{
    DeveloperAttribute[] MyAttributes =
        (DeveloperAttribute[]) Attribute.GetCustomAttributes(t, typeof
(DeveloperAttribute));

    if (MyAttributes.Length == 0)
    {
        Console.WriteLine("The attribute was not found.");
    }
    else
    {
        for (int i = 0 ; i < MyAttributes.Length ; i++)
        {
            // Get the Name value.
            Console.WriteLine("The Name Attribute is: {0}." ,
MyAttributes[i].Name);
            // Get the Level value.
            Console.WriteLine("The Level Attribute is: {0}." ,
MyAttributes[i].Level);
            // Get the Reviewed value.
            Console.WriteLine("The Reviewed Attribute is: {0}." ,
MyAttributes[i].Reviewed);
```

```
        }
    }
}
```

If no attributes are found, this code alerts the user. Otherwise, the information contained in both instances of `DeveloperAttribute` is displayed.

Retrieving Multiple Instances of an Attribute Applied to Different Scopes

The `GetCustomAttributes` and `GetCustomAttribute` methods don't search an entire class and return all instances of an attribute in that class. Rather, they search only one specified method or member at a time. If you have a class with the same attribute applied to every member and you want to retrieve the values in all the attributes applied to those members, you must supply every method or member individually to `GetCustomAttributes` and `GetCustomAttribute`.

The following code example takes a class as a parameter and searches for the `DeveloperAttribute` (defined previously) on the class level and on every individual method of that class:

C#

```
public static void GetAttribute(Type t)
{
    DeveloperAttribute att;

    // Get the class-level attributes.

    // Put the instance of the attribute on the class level in the att
    object.
    att = (DeveloperAttribute) Attribute.GetCustomAttribute (t, typeof
    (DeveloperAttribute));

    if (att == null)
    {
        Console.WriteLine("No attribute in class {0}.\n", t.ToString());
    }
    else
    {
        Console.WriteLine("The Name Attribute on the class level is: {0}.", att.Name);
        Console.WriteLine("The Level Attribute on the class level is: {0}.", att.Level);
        Console.WriteLine("The Reviewed Attribute on the class level is:
{0}.\n", att.Reviewed);
    }
}
```

```

// Get the method-level attributes.

// Get all methods in this class, and put them
// in an array of System.Reflection.MemberInfo objects.
MethodInfo[] MyMethodInfo = t.GetMethods();

// Loop through all methods in this class that are in the
// MyMethodInfo array.
for (int i = 0; i < MyMethodInfo.Length; i++)
{
    att = (DeveloperAttribute)
Attribute.GetCustomAttribute(MyMethodInfo[i], typeof(DeveloperAttribute));
    if (att == null)
    {
        Console.WriteLine("No attribute in member function {0}.\n" ,
MyMethodInfo[i].ToString());
    }
    else
    {
        Console.WriteLine("The Name Attribute for the {0} member is:
{1}.",
                           MyMethodInfo[i].ToString(), att.Name);
        Console.WriteLine("The Level Attribute for the {0} member is:
{1}.",
                           MyMethodInfo[i].ToString(), att.Level);
        Console.WriteLine("The Reviewed Attribute for the {0} member is:
{1}.\n",
                           MyMethodInfo[i].ToString(), att.Reviewed);
    }
}
}

```

If no instances of the `DeveloperAttribute` are found on the method level or class level, the `GetAttribute` method notifies the user that no attributes were found and displays the name of the method or class that doesn't contain the attribute. If an attribute is found, the console displays the `Name`, `Level`, and `Reviewed` fields.

You can use the members of the `Type` class to get the individual methods and members in the passed class. This example first queries the `Type` object to get attribute information for the class level. Next, it uses `Type.GetMethods` to place instances of all methods into an array of `System.Reflection.MemberInfo` objects to retrieve attribute information for the method level. You can also use the `Type.GetProperties` method to check for attributes on the property level or `Type.GetConstructors` to check for attributes on the constructor level.

See also

- [System.Type](#)
- [Attribute.GetCustomAttribute](#)
- [Attribute.GetCustomAttributes](#)
- [Attributes](#)

 **Collaborate with us on GitHub**

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

Memory- and span-related types

Article • 09/21/2023

Starting with .NET Core 2.1, .NET includes a number of interrelated types that represent a contiguous, strongly typed region of arbitrary memory. These include:

- [System.Span<T>](#), a type that is used to access a contiguous region of memory. A `Span<T>` instance can be backed by an array of type `T`, a buffer allocated with `stackalloc`, or a pointer to unmanaged memory. Because it has to be allocated on the stack, it has a number of restrictions. For example, a field in a class cannot be of type `Span<T>`, nor can span be used in asynchronous operations.
- [System.ReadOnlySpan<T>](#), an immutable version of the `Span<T>` structure. Instances can be also backed by a [String](#).
- [System.Memory<T>](#), a wrapper over a contiguous region of memory. A `Memory<T>` instance can be backed by an array of type `T` or a memory manager. As it can be stored on the managed heap, `Memory<T>` has none of the limitations of `Span<T>`.
- [System.ReadOnlyMemory<T>](#), an immutable version of the `Memory<T>` structure. Instances can also be backed by a [String](#).
- [System.Buffers.MemoryPool<T>](#), which allocates strongly typed blocks of memory from a memory pool to an owner. [IMemoryOwner<T>](#) instances can be rented from the pool by calling `MemoryPool<T>.Rent` and released back to the pool by calling `MemoryPool<T>.Dispose()`.
- [System.Buffers.IMemoryOwner<T>](#), which represents the owner of a block of memory and controls its lifetime management.
- [MemoryManager<T>](#), an abstract base class that can be used to replace the implementation of `Memory<T>` so that `Memory<T>` can be backed by additional types, such as safe handles. `MemoryManager<T>` is intended for advanced scenarios.
- [ArraySegment<T>](#), a wrapper for a particular number of array elements starting at a particular index.
- [System.MemoryExtensions](#), a collection of extension methods for converting strings, arrays, and array segments to `Memory<T>` blocks.

`System.Span<T>`, `System.Memory<T>`, and their readonly counterparts are designed to allow the creation of algorithms that avoid copying memory or allocating on the managed heap more than necessary. Creating them (either via `slice` or their constructors) does not involve duplicating the underlying buffers: only the relevant references and offsets, which represent the "view" of the wrapped memory, are updated.

ⓘ Note

For earlier frameworks, `Span<T>` and `Memory<T>` are available in the [System.Memory NuGet package](#).

For more information, see the [System.Buffers](#) namespace.

Working with memory and span

Because the memory- and span-related types are typically used to store data in a processing pipeline, it is important that developers follow a set of best practices when using `Span<T>`, `Memory<T>`, and related types. These best practices are documented in [Memory<T> and Span<T> usage guidelines](#).

See also

- [System.Memory<T>](#)
- [System.ReadOnlyMemory<T>](#)
- [System.Span<T>](#)
- [System.ReadOnlySpan<T>](#)
- [System.Buffers](#)

ⓘ Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



.NET feedback

The .NET documentation is open source. Provide feedback here.

ⓘ Open a documentation issue

ⓘ Provide product feedback

Memory<T> and Span<T> usage guidelines

Article • 04/19/2023

.NET includes a number of types that represent an arbitrary contiguous region of memory. [Span<T>](#) and [ReadOnlySpan<T>](#) are lightweight memory buffers that wrap references to managed or unmanaged memory. Because these types can only be stored on the stack, they're unsuitable for scenarios such as asynchronous method calls. To address this problem, .NET 2.1 added some additional types, including [Memory<T>](#), [ReadOnlyMemory<T>](#), [IMemoryOwner<T>](#), and [MemoryPool<T>](#). Like [Span<T>](#), [Memory<T>](#) and its related types can be backed by both managed and unmanaged memory. Unlike [Span<T>](#), [Memory<T>](#) can be stored on the managed heap.

Both [Span<T>](#) and [Memory<T>](#) are wrappers over buffers of structured data that can be used in pipelines. That is, they're designed so that some or all of the data can be efficiently passed to components in the pipeline, which can process them and optionally modify the buffer. Because [Memory<T>](#) and its related types can be accessed by multiple components or by multiple threads, it's important to follow some standard usage guidelines to produce robust code.

Owners, consumers, and lifetime management

Buffers can be passed around between APIs and can sometimes be accessed from multiple threads, so be aware of how a buffer's lifetime is managed. There are three core concepts:

- **Ownership.** The owner of a buffer instance is responsible for lifetime management, including destroying the buffer when it's no longer in use. All buffers have a single owner. Generally the owner is the component that created the buffer or that received the buffer from a factory. Ownership can also be transferred; **Component-A** can relinquish control of the buffer to **Component-B**, at which point **Component-A** may no longer use the buffer, and **Component-B** becomes responsible for destroying the buffer when it's no longer in use.
- **Consumption.** The consumer of a buffer instance is allowed to use the buffer instance by reading from it and possibly writing to it. Buffers can have one consumer at a time unless some external synchronization mechanism is provided. The active consumer of a buffer isn't necessarily the buffer's owner.

- **Lease.** The lease is the length of time that a particular component is allowed to be the consumer of the buffer.

The following pseudo-code example illustrates these three concepts. `Buffer` in the pseudo-code represents a `Memory<T>` or `Span<T>` buffer of type `Char`. The `Main` method instantiates the buffer, calls the `WriteInt32ToBuffer` method to write the string representation of an integer to the buffer, and then calls the `DisplayBufferToConsole` method to display the value of the buffer.

C#

```
using System;

class Program
{
    // Write 'value' as a human-readable string to the output buffer.
    void WriteInt32ToBuffer(int value, Buffer buffer);

    // Display the contents of the buffer to the console.
    void DisplayBufferToConsole(Buffer buffer);

    // Application code
    static void Main()
    {
        var buffer = CreateBuffer();
        try
        {
            int value = Int32.Parse(Console.ReadLine());
            WriteInt32ToBuffer(value, buffer);
            DisplayBufferToConsole(buffer);
        }
        finally
        {
            buffer.Destroy();
        }
    }
}
```

The `Main` method creates the buffer and so is its owner. Therefore, `Main` is responsible for destroying the buffer when it's no longer in use. The pseudo-code illustrates this by calling a `Destroy` method on the buffer. (Neither `Memory<T>` nor `Span<T>` actually has a `Destroy` method. You'll see actual code examples later in this article.)

The buffer has two consumers, `WriteInt32ToBuffer` and `DisplayBufferToConsole`. There is only one consumer at a time (first `WriteInt32ToBuffer`, then `DisplayBufferToConsole`), and neither of the consumers owns the buffer. Note also that "consumer" in this context

doesn't imply a read-only view of the buffer; consumers can modify the buffer's contents, as `WriteInt32ToBuffer` does, if given a read/write view of the buffer.

The `WriteInt32ToBuffer` method has a lease on (can consume) the buffer between the start of the method call and the time the method returns. Similarly, `DisplayBufferToConsole` has a lease on the buffer while it's executing, and the lease is released when the method unwinds. (There is no API for lease management; a "lease" is a conceptual matter.)

Memory<T> and the owner/consumer model

As the [Owners, consumers, and lifetime management](#) section notes, a buffer always has an owner. .NET supports two ownership models:

- A model that supports single ownership. A buffer has a single owner for its entire lifetime.
- A model that supports ownership transfer. Ownership of a buffer can be transferred from its original owner (its creator) to another component, which then becomes responsible for the buffer's lifetime management. That owner can in turn transfer ownership to another component, and so on.

You use the `System.Buffers.IMemoryOwner<T>` interface to explicitly manage the ownership of a buffer. `IMemoryOwner<T>` supports both ownership models. The component that has an `IMemoryOwner<T>` reference owns the buffer. The following example uses an `IMemoryOwner<T>` instance to reflect the ownership of a `Memory<T>` buffer.

C#

```
using System;
using System.Buffers;

class Example
{
    static void Main()
    {
        IMemoryOwner<char> owner = MemoryPool<char>.Shared.Rent();

        Console.Write("Enter a number: ");
        try
        {
            string? s = Console.ReadLine();

            if (s is null)
                return;
        }
    }
}
```

```

        var value = Int32.Parse(s);

        var memory = owner.Memory;

        WriteInt32ToBuffer(value, memory);

        DisplayBufferToConsole(owner.Memory.Slice(0,
value.ToString().Length));
    }
    catch (FormatException)
    {
        Console.WriteLine("You did not enter a valid number.");
    }
    catch (OverflowException)
    {
        Console.WriteLine($"You entered a number less than
{Int32.MinValue:N0} or greater than {Int32.MaxValue:N0}.");
    }
    finally
    {
        owner?.Dispose();
    }
}

static void WriteInt32ToBuffer(int value, Memory<char> buffer)
{
    var strValue = value.ToString();

    var span = buffer.Span;
    for (int ctr = 0; ctr < strValue.Length; ctr++)
        span[ctr] = strValue[ctr];
}

static void DisplayBufferToConsole(Memory<char> buffer) =>
    Console.WriteLine($"Contents of the buffer: '{buffer}'");
}

```

We can also write this example with the [using statement](#):

```

C#

using System;
using System.Buffers;

class Example
{
    static void Main()
    {
        using (IMemoryOwner<char> owner = MemoryPool<char>.Shared.Rent())
        {
            Console.Write("Enter a number: ");
            try

```

```

    {
        string? s = Console.ReadLine();

        if (s is null)
            return;

        var value = Int32.Parse(s);

        var memory = owner.Memory;
        WriteInt32ToBuffer(value, memory);
        DisplayBufferToConsole(memory.Slice(0,
value.ToString().Length));
    }
    catch (FormatException)
    {
        Console.WriteLine("You did not enter a valid number.");
    }
    catch (OverflowException)
    {
        Console.WriteLine($"You entered a number less than
{Int32.MinValue:N0} or greater than {Int32.MaxValue:N0}.");
    }
}

static void WriteInt32ToBuffer(int value, Memory<char> buffer)
{
    var strValue = value.ToString();

    var span = buffer.Slice(0, strValue.Length).Span;
    strValue.AsSpan().CopyTo(span);
}

static void DisplayBufferToConsole(Memory<char> buffer) =>
    Console.WriteLine($"Contents of the buffer: '{buffer}'");
}

```

In this code:

- The `Main` method holds the reference to the `IMemoryOwner<T>` instance, so the `Main` method is the owner of the buffer.
- The `WriteInt32ToBuffer` and `DisplayBufferToConsole` methods accept `Memory<T>` as a public API. Therefore, they are consumers of the buffer. These methods consume the buffer one at a time.

Although the `WriteInt32ToBuffer` method is intended to write a value to the buffer, the `DisplayBufferToConsole` method isn't intended to. To reflect this, it could have accepted an argument of type `ReadOnlyMemory<T>`. For more information on

`ReadOnlyMemory<T>`, see Rule #2: Use `ReadOnlySpan<T>` or `ReadOnlyMemory<T>` if the buffer should be read-only.

"Ownerless" `Memory<T>` instances

You can create a `Memory<T>` instance without using `IMemoryOwner<T>`. In this case, ownership of the buffer is implicit rather than explicit, and only the single-owner model is supported. You can do this by:

- Calling one of the `Memory<T>` constructors directly, passing in a `T[]`, as the following example does.
- Calling the `String.AsMemory` extension method to produce a `ReadOnlyMemory<char>` instance.

C#

```
using System;

class Example
{
    static void Main()
    {
        Memory<char> memory = new char[64];

        Console.Write("Enter a number: ");
        string? s = Console.ReadLine();

        if (s is null)
            return;

        var value = Int32.Parse(s);

        WriteInt32ToBuffer(value, memory);
        DisplayBufferToConsole(memory);
    }

    static void WriteInt32ToBuffer(int value, Memory<char> buffer)
    {
        var strValue = value.ToString();
        strValue.AsSpan().CopyTo(buffer.Slice(0, strValue.Length).Span);
    }

    static void DisplayBufferToConsole(Memory<char> buffer) =>
        Console.WriteLine($"Contents of the buffer: '{buffer}'");
}
```

The method that initially creates the `Memory<T>` instance is the implicit owner of the buffer. Ownership cannot be transferred to any other component because there is no

`IMemoryOwner<T>` instance to facilitate the transfer. (As an alternative, you can also imagine that the runtime's garbage collector owns the buffer, and all methods just consume the buffer.)

Usage guidelines

Because a memory block is owned but is intended to be passed to multiple components, some of which may operate upon a particular memory block simultaneously, it's important to establish guidelines for using both `Memory<T>` and `Span<T>`. Guidelines are necessary because it's possible for a component to:

- Retain a reference to a memory block after its owner has released it.
- Operate on a buffer at the same time that another component is operating on it, in the process corrupting the data in the buffer.
- While the stack-allocated nature of `Span<T>` optimizes performance and makes `Span<T>` the preferred type for operating on a memory block, it also subjects `Span<T>` to some major restrictions. It's important to know when to use a `Span<T>` and when to use `Memory<T>`.

The following are our recommendations for successfully using `Memory<T>` and its related types. Guidance that applies to `Memory<T>` and `Span<T>` also applies to `ReadOnlyMemory<T>` and `ReadOnlySpan<T>` unless noted otherwise.

Rule #1: For a synchronous API, use `Span<T>` instead of `Memory<T>` as a parameter if possible.

`Span<T>` is more versatile than `Memory<T>` and can represent a wider variety of contiguous memory buffers. `Span<T>` also offers better performance than `Memory<T>`. Finally, you can use the `Memory<T>.Span` property to convert a `Memory<T>` instance to a `Span<T>`, although `Span<T>`-to-`Memory<T>` conversion isn't possible. So if your callers happen to have a `Memory<T>` instance, they'll be able to call your methods with `Span<T>` parameters anyway.

Using a parameter of type `Span<T>` instead of type `Memory<T>` also helps you write a correct consuming method implementation. You'll automatically get compile-time checks to ensure that you're not attempting to access the buffer beyond your method's lease (more on this later).

Sometimes, you'll have to use a `Memory<T>` parameter instead of a `Span<T>` parameter, even if you're fully synchronous. Perhaps an API that you depend on accepts

only `Memory<T>` arguments. This is fine, but be aware of the tradeoffs involved when using `Memory<T>` synchronously.

Rule #2: Use `ReadOnlySpan<T>` or `ReadOnlyMemory<T>` if the buffer should be read-only.

In the earlier examples, the `DisplayBufferToConsole` method only reads from the buffer; it doesn't modify the contents of the buffer. The method signature should be changed to the following.

C#

```
void DisplayBufferToConsole(ReadOnlyMemory<char> buffer);
```

In fact, if we combine this rule and Rule #1, we can do even better and rewrite the method signature as follows:

C#

```
void DisplayBufferToConsole(ReadOnlySpan<char> buffer);
```

The `DisplayBufferToConsole` method now works with virtually every buffer type imaginable: `T[]`, storage allocated with `stackalloc`, and so on. You can even pass a `String` directly into it! For more information, see GitHub issue [dotnet/docs #25551](#).

Rule #3: If your method accepts `Memory<T>` and returns `void`, you must not use the `Memory<T>` instance after your method returns.

This relates to the "lease" concept mentioned earlier. A void-returning method's lease on the `Memory<T>` instance begins when the method is entered, and it ends when the method exits. Consider the following example, which calls `Log` in a loop based on input from the console.

C#

```
using System;
using System.Buffers;

public class Example
{
    // implementation provided by third party
    static extern void Log(ReadOnlyMemory<char> message);

    // user code
    public static void Main()
    {
```

```

        using (var owner = MemoryPool<char>.Shared.Rent())
    {
        var memory = owner.Memory;
        var span = memory.Span;
        while (true)
        {
            string? s = Console.ReadLine();

            if (s is null)
                return;

            int value = Int32.Parse(s);
            if (value < 0)
                return;

            int numCharsWritten = ToBuffer(value, span);
            Log(memory.Slice(0, numCharsWritten));
        }
    }

private static int ToBuffer(int value, Span<char> span)
{
    string strValue = value.ToString();
    int length = strValue.Length;
    strValue.AsSpan().CopyTo(span.Slice(0, length));
    return length;
}
}

```

If `Log` is a fully synchronous method, this code will behave as expected because there is only one active consumer of the memory instance at any given time. But imagine instead that `Log` has this implementation.

C#

```

// !!! INCORRECT IMPLEMENTATION !!!
static void Log(ReadOnlyMemory<char> message)
{
    // Run in background so that we don't block the main thread while
    performing IO.
    Task.Run(() =>
    {
        StreamWriter sw = File.AppendText(@"..\input-numbers.dat");
        sw.WriteLine(message);
    });
}

```

In this implementation, `Log` violates its lease because it still attempts to use the `Memory<T>` instance in the background after the original method has returned. The

`Main` method could mutate the buffer while `Log` attempts to read from it, which could result in data corruption.

There are several ways to resolve this:

- The `Log` method can return a `Task` instead of `void`, as the following implementation of the `Log` method does.

C#

```
// An acceptable implementation.
static Task Log(ReadOnlyMemory<char> message)
{
    // Run in the background so that we don't block the main thread
    // while performing IO.
    return Task.Run(() => {
        StreamWriter sw = File.AppendText(@".\input-numbers.dat");
        sw.WriteLine(message);
        sw.Flush();
    });
}
```

- `Log` can instead be implemented as follows:

C#

```
// An acceptable implementation.
static void Log(ReadOnlyMemory<char> message)
{
    string defensiveCopy = message.ToString();
    // Run in the background so that we don't block the main thread
    // while performing IO.
    Task.Run(() =>
    {
        StreamWriter sw = File.AppendText(@".\input-numbers.dat");
        sw.WriteLine(defensiveCopy);
        sw.Flush();
    });
}
```

Rule #4: If your method accepts a `Memory<T>` and returns a `Task`, you must not use the `Memory<T>` instance after the `Task` transitions to a terminal state.

This is just the async variant of Rule #3. The `Log` method from the earlier example can be written as follows to comply with this rule:

C#

```
// An acceptable implementation.
static Task Log(ReadOnlyMemory<char> message)
{
    // Run in the background so that we don't block the main thread while
    // performing IO.
    return Task.Run(() =>
    {
        StreamWriter sw = File.AppendText(@".\input-numbers.dat");
        sw.WriteLine(message);
        sw.Flush();
    });
}
```

Here, "terminal state" means that the task transitions to a completed, faulted, or canceled state. In other words, "terminal state" means "anything that would cause await to throw or to continue execution."

This guidance applies to methods that return `Task`, `Task<TResult>`, `ValueTask<TResult>`, or any similar type.

Rule #5: If your constructor accepts `Memory<T>` as a parameter, instance methods on the constructed object are assumed to be consumers of the `Memory<T>` instance.

Consider the following example:

C#

```
class OddValueExtractor
{
    public OddValueExtractor(ReadOnlyMemory<int> input);
    public bool TryReadNextOddValue(out int value);
}

void PrintAllOddValues(ReadOnlyMemory<int> input)
{
    var extractor = new OddValueExtractor(input);
    while (extractor.TryReadNextOddValue(out int value))
    {
        Console.WriteLine(value);
    }
}
```

Here, the `OddValueExtractor` constructor accepts a `ReadOnlyMemory<int>` as a constructor parameter, so the constructor itself is a consumer of the `ReadOnlyMemory<int>` instance, and all instance methods on the returned value are also consumers of the original `ReadOnlyMemory<int>` instance. This means that

`TryReadNextOddValue` consumes the `ReadOnlyMemory<int>` instance, even though the instance isn't passed directly to the `TryReadNextOddValue` method.

Rule #6: If you have a settable `Memory<T>`-typed property (or an equivalent instance method) on your type, instance methods on that object are assumed to be consumers of the `Memory<T>` instance.

This is really just a variant of Rule #5. This rule exists because property setters or equivalent methods are assumed to capture and persist their inputs, so instance methods on the same object may utilize the captured state.

The following example triggers this rule:

C#

```
class Person
{
    // Settable property.
    public Memory<char> FirstName { get; set; }

    // alternatively, equivalent "setter" method
    public SetFirstName(Memory<char> value);

    // alternatively, a public settable field
    public Memory<char> FirstName;
}
```

Rule #7: If you have an `IMemoryOwner<T>` reference, you must at some point dispose of it or transfer its ownership (but not both).

Since a `Memory<T>` instance may be backed by either managed or unmanaged memory, the owner must call `Dispose` on `IMemoryOwner<T>` when work performed on the `Memory<T>` instance is complete. Alternatively, the owner may transfer ownership of the `IMemoryOwner<T>` instance to a different component, at which point the acquiring component becomes responsible for calling `Dispose` at the appropriate time (more on this later).

Failure to call the `Dispose` method on an `IMemoryOwner<T>` instance may lead to unmanaged memory leaks or other performance degradation.

This rule also applies to code that calls factory methods like `MemoryPool<T>.Rent`. The caller becomes the owner of the returned `IMemoryOwner<T>` and is responsible for disposing of the instance when finished.

Rule #8: If you have an `IMemoryOwner<T>` parameter in your API surface, you are accepting ownership of that instance.

Accepting an instance of this type signals that your component intends to take ownership of this instance. Your component becomes responsible for proper disposal according to Rule #7.

Any component that transfers ownership of the `IMemoryOwner<T>` instance to a different component should no longer use that instance after the method call completes.

ⓘ Important

If your constructor accepts `IMemoryOwner<T>` as a parameter, its type should implement `IDisposable`, and your `Dispose` method should call `Dispose` on the `IMemoryOwner<T>` object.

Rule #9: If you're wrapping a synchronous p/invoke method, your API should accept `Span<T>` as a parameter.

According to Rule #1, `Span<T>` is generally the correct type to use for synchronous APIs. You can pin `Span<T>` instances via the `fixed` keyword, as in the following example.

C#

```
using System.Runtime.InteropServices;

[DllImport(...)]
private static extern unsafe int ExportedMethod(byte* pbData, int cbData);

public unsafe int ManagedWrapper(Span<byte> data)
{
    fixed (byte* pbData = &MemoryMarshal.GetReference(data))
    {
        int retVal = ExportedMethod(pbData, data.Length);

        /* error checking retVal goes here */

        return retVal;
    }
}
```

In the previous example, `pbData` can be null if, for example, the input span is empty. If the exported method absolutely requires that `pbData` be non-null, even if `cbData` is 0, the method can be implemented as follows:

C#

```

public unsafe int ManagedWrapper(Span<byte> data)
{
    fixed (byte* pbData = &MemoryMarshal.GetReference(data))
    {
        byte dummy = 0;
        int retVal = ExportedMethod((pbData != null) ? pbData : &dummy,
data.Length);

        /* error checking retVal goes here */

        return retVal;
    }
}

```

Rule #10: If you're wrapping an asynchronous p/invoke method, your API should accept Memory<T> as a parameter.

Since you cannot use the `fixed` keyword across asynchronous operations, you use the `Memory<T>.Pin` method to pin `Memory<T>` instances, regardless of the kind of contiguous memory the instance represents. The following example shows how to use this API to perform an asynchronous p/invoke call.

C#

```

using System.Runtime.InteropServices;

[UnmanagedFunctionPointer(...)]
private delegate void OnCompletedCallback(IntPtr state, int result);

[DllImport(...)]
private static extern unsafe int ExportedAsyncMethod(byte* pbData, int
cbData, IntPtr pState, IntPtr lpfnOnCompletedCallback);

private static readonly IntPtr _callbackPtr =
GetCompletionCallbackPointer();

public unsafe Task<int> ManagedWrapperAsync(Memory<byte> data)
{
    // setup
    var tcs = new TaskCompletionSource<int>();
    var state = new MyCompletedCallbackState
    {
        Tcs = tcs
    };
    var pState = (IntPtr)GCHandle.Alloc(state);

    var memoryHandle = data.Pin();
    state.MemoryHandle = memoryHandle;

    // make the call
    int result;

```

```
try
{
    result = ExportedAsyncMethod((byte*)memoryHandle.Pointer,
data.Length, pState, _callbackPtr);
}
catch
{
    ((GCHandle)pState).Free(); // cleanup since callback won't be
invoked
    memoryHandle.Dispose();
    throw;
}

if (result != PENDING)
{
    // Operation completed synchronously; invoke callback manually
    // for result processing and cleanup.
    MyCompletedCallbackImplementation(pState, result);
}

return tcs.Task;
}

private static void MyCompletedCallbackImplementation(IntPtr state, int
result)
{
    GCHandle handle = (GCHandle)state;
    var actualState = (MyCompletedCallbackState)(handle.Target);
    handle.Free();
    actualState.MemoryHandle.Dispose();

    /* error checking result goes here */

    if (error)
    {
        actualState.Tcs.SetException(...);
    }
    else
    {
        actualState.Tcs.SetResult(result);
    }
}

private static IntPtr GetCompletionCallbackPointer()
{
    OnCompletedCallback callback = MyCompletedCallbackImplementation;
    GCHandle.Alloc(callback); // keep alive for lifetime of application
    return Marshal.GetFunctionPointerForDelegate(callback);
}

private class MyCompletedCallbackState
{
    public TaskCompletionSource<int> Tcs;
    public MemoryHandle MemoryHandle;
}
```

See also

- [System.Memory<T>](#)
- [System.Buffers.IMemoryOwner<T>](#)
- [System.Span<T>](#)

 Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

Use SIMD-accelerated numeric types

Article • 06/07/2022

SIMD (Single instruction, multiple data) provides hardware support for performing an operation on multiple pieces of data, in parallel, using a single instruction. In .NET, there's set of SIMD-accelerated types under the [System.Numerics](#) namespace. SIMD operations can be parallelized at the hardware level. That increases the throughput of the vectorized computations, which are common in mathematical, scientific, and graphics apps.

.NET SIMD-accelerated types

The .NET SIMD-accelerated types include the following types:

- The [Vector2](#), [Vector3](#), and [Vector4](#) types, which represent vectors with 2, 3, and 4 [Single](#) values.
- Two matrix types, [Matrix3x2](#), which represents a 3x2 matrix, and [Matrix4x4](#), which represents a 4x4 matrix of [Single](#) values.
- The [Plane](#) type, which represents a plane in three-dimensional space using [Single](#) values.
- The [Quaternion](#) type, which represents a vector that is used to encode three-dimensional physical rotations using [Single](#) values.
- The [Vector<T>](#) type, which represents a vector of a specified numeric type and provides a broad set of operators that benefit from SIMD support. The count of a [Vector<T>](#) instance is fixed for the lifetime of an application, but its value [Vector<T>.Count](#) depends on the CPU of the machine running the code.

Note

The [Vector<T>](#) type is not included in the .NET Framework. You must install the [System.Numerics.Vectors](#) NuGet package to get access to this type.

The SIMD-accelerated types are implemented in such a way that they can be used with non-SIMD-accelerated hardware or JIT compilers. To take advantage of SIMD instructions, your 64-bit apps must be run by the runtime that uses the [RyuJIT](#) compiler. A [RyuJIT](#) compiler is included in .NET Core and in .NET Framework 4.6 and later. SIMD support is only provided when targeting 64-bit processors.

How to use SIMD?

Before executing custom SIMD algorithms, it's possible to check if the host machine supports SIMD by using [Vector.IsHardwareAccelerated](#), which returns a [Boolean](#). This doesn't guarantee that SIMD-acceleration is enabled for a specific type, but is an indicator that it's supported by some types.

Simple Vectors

The most primitive SIMD-accelerated types in .NET are [Vector2](#), [Vector3](#), and [Vector4](#) types, which represent vectors with 2, 3, and 4 [Single](#) values. The example below uses [Vector2](#) to add two vectors.

C#

```
var v1 = new Vector2(0.1f, 0.2f);
var v2 = new Vector2(1.1f, 2.2f);
var vResult = v1 + v2;
```

It's also possible to use .NET vectors to calculate other mathematical properties of vectors such as [Dot product](#), [Transform](#), [Clamp](#) and so on.

C#

```
var v1 = new Vector2(0.1f, 0.2f);
var v2 = new Vector2(1.1f, 2.2f);
var vResult1 = Vector2.Dot(v1, v2);
var vResult2 = Vector2.Distance(v1, v2);
var vResult3 = Vector2.Clamp(v1, Vector2.Zero, Vector2.One);
```

Matrix

[Matrix3x2](#), which represents a 3x2 matrix, and [Matrix4x4](#), which represents a 4x4 matrix. Can be used for matrix-related calculations. The example below demonstrates multiplication of a matrix to its correspondent transpose matrix using SIMD.

C#

```
var m1 = new Matrix4x4(
    1.1f, 1.2f, 1.3f, 1.4f,
    2.1f, 2.2f, 3.3f, 4.4f,
    3.1f, 3.2f, 3.3f, 3.4f,
    4.1f, 4.2f, 4.3f, 4.4f);
```

```
var m2 = Matrix4x4.Transpose(m1);
var mResult = Matrix4x4.Multiply(m1, m2);
```

Vector<T>

The `Vector<T>` gives the ability to use longer vectors. The count of a `Vector<T>` instance is fixed, but its value `Vector<T>.Count` depends on the CPU of the machine running the code.

The following example demonstrates how to calculate the element-wise sum of two arrays using `Vector<T>`.

C#

```
double[] Sum(double[] left, double[] right)
{
    if (left is null)
    {
        throw new ArgumentNullException(nameof(left));
    }

    if (right is null)
    {
        throw new ArgumentNullException(nameof(right));
    }

    if (left.Length != right.Length)
    {
        throw new ArgumentException($"{nameof(left)} and {nameof(right)} are
not the same length");
    }

    int length = left.Length;
    double[] result = new double[length];

    // Get the number of elements that can't be processed in the vector
    // NOTE: Vector<T>.Count is a JIT time constant and will get optimized
    // accordingly
    int remaining = length % Vector<double>.Count;

    for (int i = 0; i < length - remaining; i += Vector<double>.Count)
    {
        var v1 = new Vector<double>(left, i);
        var v2 = new Vector<double>(right, i);
        (v1 + v2).CopyTo(result, i);
    }

    for (int i = length - remaining; i < length; i++)
    {
        result[i] = left[i] + right[i];
    }
}
```

```
    }

    return result;
}
```

Remarks

SIMD is more likely to remove one bottleneck and expose the next, for example memory throughput. In general the performance benefit of using SIMD varies depending on the specific scenario, and in some cases it can even perform worse than simpler non-SIMD equivalent code.

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

Value tuples

Article • 01/08/2024

A value tuple is a data structure that has a specific number and sequence of values. .NET provides the following built-in value tuple types:

- The `ValueTuple<T1>` structure represents a value tuple that has one element.
- The `ValueTuple<T1,T2>` structure represents a value tuple that has two elements.-
- The `ValueTuple<T1,T2,T3>` structure represents a value tuple that has three elements.
- The `ValueTuple<T1,T2,T3,T4>` structure represents a value tuple that has four elements.
- The `ValueTuple<T1,T2,T3,T4,T5>` structure represents a value tuple that has five elements.
- The `ValueTuple<T1,T2,T3,T4,T5,T6>` structure represents a value tuple that has six elements.
- The `ValueTuple<T1,T2,T3,T4,T5,T6,T7>` structure represents a value tuple that has seven elements.
- The `ValueTuple<T1,T2,T3,T4,T5,T6,T7,TRest>` structure represents a value tuple that has eight or more elements.

The value tuple types differ from the tuple types (such as `Tuple<T1,T2>`) as follows:

- They are structures (value types) rather than classes (reference types).
- Members such as `Item1` and `Item2` are fields rather than properties.
- Their fields are mutable rather than read-only.

The value tuple types provide the runtime implementation that supports [tuples in C#](#) and struct tuples in F#. In addition to creating a `ValueTuple<T1,T2>` instance by using language syntax, you can call the `Create` factory method.

See also

- [Tuple types \(C# reference\)](#)

 Collaborate with us on
GitHub

The source for this content can
be found on GitHub, where you



.NET feedback

.NET is an open source project.
Select a link to provide feedback:

can also create and review issues and pull requests. For more information, see [our contributor guide](#).

 Open a documentation issue

 Provide product feedback

Runtime libraries overview

Article • 01/10/2024

The .NET runtime has an expansive standard set of class libraries, known as [runtime libraries](#), [framework libraries](#), or the [base class library \(BCL\)](#). In addition, there are extensions to the runtime libraries, provided in NuGet packages.

These libraries provide implementations for many general and app-specific types, algorithms, and utility functionality.

Runtime libraries

These libraries provide the foundational types and utility functionality and are the base of all other .NET class libraries. An example is the [System.String](#) class, which provides APIs for working with strings. Another example is the [serialization libraries](#).

Extensions to the runtime libraries

Some libraries are provided in NuGet packages rather than included in the runtime's [shared framework](#). For example:

 Expand table

| Conceptual content | NuGet package |
|----------------------|--|
| Configuration | Microsoft.Extensions.Configuration ↗ |
| Dependency injection | Microsoft.Extensions.DependencyInjection ↗ |
| File globbing | Microsoft.Extensions.FileSystemGlobbing ↗ |
| Generic Host | Microsoft.Extensions.Hosting ↗ |
| HTTP | [†] Microsoft.Extensions.Http ↗ |
| Localization | Microsoft.Extensions.Localization ↗ |
| Logging | Microsoft.Extensions.Logging ↗ |

[†] For some target frameworks, including `net6.0`, these libraries are part of the shared framework and don't need to be installed separately.

See also

- [Introduction to .NET](#)
- [Install .NET SDK or runtime](#)
- [Select the installed .NET SDK or runtime version to use](#)
- [Publish framework-dependent apps](#)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

.NET is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

Overview: How to format numbers, dates, enums, and other types in .NET

Article • 10/28/2022

Formatting is the process of converting an instance of a class or structure, or an enumeration value, to a string representation. The purpose is to display the resulting string to users or to deserialize it later to restore the original data type. This article introduces the formatting mechanisms that .NET provides.

ⓘ Note

Parsing is the inverse of formatting. A parsing operation creates an instance of a data type from its string representation. For more information, see [Parsing Strings](#). For information about serialization and deserialization, see [Serialization in .NET](#).

The basic mechanism for formatting is the default implementation of the [Object.ToString](#) method, which is discussed in the [Default Formatting Using the ToString Method](#) section later in this topic. However, .NET provides several ways to modify and extend its default formatting support. These include the following:

- Overriding the [Object.ToString](#) method to define a custom string representation of an object's value. For more information, see the [Override the ToString Method](#) section later in this topic.
- Defining format specifiers that enable the string representation of an object's value to take multiple forms. For example, the "X" format specifier in the following statement converts an integer to the string representation of a hexadecimal value.

C#

```
int integerValue = 60312;
Console.WriteLine(integerValue.ToString("X")); // Displays EB98.
```

For more information about format specifiers, see the [ToString Method and Format Strings](#) section.

- Using format providers to implement the formatting conventions of a specific culture. For example, the following statement displays a currency value by using the formatting conventions of the en-US culture.

C#

```
double cost = 1632.54;
Console.WriteLine(cost.ToString("C",
    new System.Globalization.CultureInfo("en-US")));
// The example displays the following output:
//      $1,632.54
```

For more information about formatting with format providers, see the [Format Providers](#) section.

- Implementing the [IFormattable](#) interface to support both string conversion with the [Convert](#) class and composite formatting. For more information, see the [IFormattable Interface](#) section.
- Using composite formatting to embed the string representation of a value in a larger string. For more information, see the [Composite Formatting](#) section.
- Using string interpolation, a more readable syntax to embed the string representation of a value in a larger string. For more information, see [String interpolation](#).
- Implementing [ICustomFormatter](#) and [IFormatProvider](#) to provide a complete custom formatting solution. For more information, see the [Custom Formatting with ICustomFormatter](#) section.

The following sections examine these methods for converting an object to its string representation.

Default formatting using the `ToString` method

Every type that is derived from [System.Object](#) automatically inherits a parameterless `ToString` method, which returns the name of the type by default. The following example illustrates the default `ToString` method. It defines a class named `Automobile` that has no implementation. When the class is instantiated and its `ToString` method is called, it displays its type name. Note that the `ToString` method is not explicitly called in the example. The `Console.WriteLine(Object)` method implicitly calls the `ToString` method of the object passed to it as an argument.

C#

```
using System;

public class Automobile
{
    // No implementation. All members are inherited from Object.
```

```
}

public class Example9
{
    public static void Main()
    {
        Automobile firstAuto = new Automobile();
        Console.WriteLine(firstAuto);
    }
}
// The example displays the following output:
//      Automobile
```

⚠ Warning

Starting with Windows 8.1, the Windows Runtime includes an **IStringable** interface with a single method, **IStringable.ToString**, which provides default formatting support. However, we recommend that managed types do not implement the **IStringable** interface. For more information, see "The Windows Runtime and the **IStringable** Interface" section on the **Object.ToString** reference page.

Because all types other than interfaces are derived from **Object**, this functionality is automatically provided to your custom classes or structures. However, the functionality offered by the default **ToString** method, is limited: Although it identifies the type, it fails to provide any information about an instance of the type. To provide a string representation of an object that provides information about that object, you must override the **ToString** method.

ⓘ Note

Structures inherit from **ValueType**, which in turn is derived from **Object**. Although **ValueType** overrides **Object.ToString**, its implementation is identical.

Override the **ToString** method

Displaying the name of a type is often of limited use and does not allow consumers of your types to differentiate one instance from another. However, you can override the **ToString** method to provide a more useful representation of an object's value. The following example defines a **Temperature** object and overrides its **ToString** method to display the temperature in degrees Celsius.

```

public class Temperature
{
    private decimal temp;

    public Temperature(decimal temperature)
    {
        this.temp = temperature;
    }

    public override string ToString()
    {
        return this.temp.ToString("N1") + "°C";
    }
}

public class Example12
{
    public static void Main()
    {
        Temperature currentTemperature = new Temperature(23.6m);
        Console.WriteLine($"The current temperature is
{currentTemperature}");
    }
}
// The example displays the following output:
//      The current temperature is 23.6°C.

```

In .NET, the `ToString` method of each primitive value type has been overridden to display the object's value instead of its name. The following table shows the override for each primitive type. Note that most of the overridden methods call another overload of the `ToString` method and pass it the "G" format specifier, which defines the general format for its type, and an `IFormatProvider` object that represents the current culture.

| Type | <code>ToString</code> override |
|----------|--|
| Boolean | Returns either <code>Boolean.TrueString</code> or <code>Boolean.FalseString</code> . |
| Byte | Calls <code>Byte.ToString("G", NumberFormatInfo.CurrentInfo)</code> to format the <code>Byte</code> value for the current culture. |
| Char | Returns the character as a string. |
| DateTime | Calls <code>DateTime.ToString("G", DatetimeFormatInfo.CurrentInfo)</code> to format the date and time value for the current culture. |
| Decimal | Calls <code>Decimal.ToString("G", NumberFormatInfo.CurrentInfo)</code> to format the <code>Decimal</code> value for the current culture. |
| Double | Calls <code>Double.ToString("G", NumberFormatInfo.CurrentInfo)</code> to format the <code>Double</code> |

| Type | ToString override |
|--------|--|
| | value for the current culture. |
| Int16 | Calls <code>Int16.ToString("G", NumberFormatInfo.CurrentInfo)</code> to format the <code>Int16</code> value for the current culture. |
| Int32 | Calls <code>Int32.ToString("G", NumberFormatInfo.CurrentInfo)</code> to format the <code>Int32</code> value for the current culture. |
| Int64 | Calls <code>Int64.ToString("G", NumberFormatInfo.CurrentInfo)</code> to format the <code>Int64</code> value for the current culture. |
| SByte | Calls <code>SByte.ToString("G", NumberFormatInfo.CurrentInfo)</code> to format the <code>SByte</code> value for the current culture. |
| Single | Calls <code>Single.ToString("G", NumberFormatInfo.CurrentInfo)</code> to format the <code>Single</code> value for the current culture. |
| UInt16 | Calls <code>UInt16.ToString("G", NumberFormatInfo.CurrentInfo)</code> to format the <code>UInt16</code> value for the current culture. |
| UInt32 | Calls <code>UInt32.ToString("G", NumberFormatInfo.CurrentInfo)</code> to format the <code>UInt32</code> value for the current culture. |
| UInt64 | Calls <code>UInt64.ToString("G", NumberFormatInfo.CurrentInfo)</code> to format the <code>UInt64</code> value for the current culture. |

The `ToString` method and format strings

Relying on the default `ToString` method or overriding `ToString` is appropriate when an object has a single string representation. However, the value of an object often has multiple representations. For example, a temperature can be expressed in degrees Fahrenheit, degrees Celsius, or kelvins. Similarly, the integer value 10 can be represented in numerous ways, including 10, 10.0, 1.0e01, or \$10.00.

To enable a single value to have multiple string representations, .NET uses format strings. A format string is a string that contains one or more predefined format specifiers, which are single characters or groups of characters that define how the `ToString` method should format its output. The format string is then passed as a parameter to the object's `ToString` method and determines how the string representation of that object's value should appear.

All numeric types, date and time types, and enumeration types in .NET support a predefined set of format specifiers. You can also use format strings to define multiple string representations of your application-defined data types.

Standard format strings

A standard format string contains a single format specifier, which is an alphabetic character that defines the string representation of the object to which it is applied, along with an optional precision specifier that affects how many digits are displayed in the result string. If the precision specifier is omitted or is not supported, a standard format specifier is equivalent to a standard format string.

.NET defines a set of standard format specifiers for all numeric types, all date and time types, and all enumeration types. For example, each of these categories supports a "G" standard format specifier, which defines a general string representation of a value of that type.

Standard format strings for enumeration types directly control the string representation of a value. The format strings passed to an enumeration value's `ToString` method determine whether the value is displayed using its string name (the "G" and "F" format specifiers), its underlying integral value (the "D" format specifier), or its hexadecimal value (the "X" format specifier). The following example illustrates the use of standard format strings to format a `DayOfWeek` enumeration value.

C#

```
DayOfWeek thisDay = DayOfWeek.Monday;
string[] formatStrings = {"G", "F", "D", "X"};

foreach (string formatString in formatStrings)
    Console.WriteLine(thisDay.ToString(formatString));
// The example displays the following output:
//      Monday
//      Monday
//      1
//      00000001
```

For information about enumeration format strings, see [Enumeration Format Strings](#).

Standard format strings for numeric types usually define a result string whose precise appearance is controlled by one or more property values. For example, the "C" format specifier formats a number as a currency value. When you call the `ToString` method with the "C" format specifier as the only parameter, the following property values from the current culture's `NumberFormatInfo` object are used to define the string representation of the numeric value:

- The `CurrencySymbol` property, which specifies the current culture's currency symbol.

- The [CurrencyNegativePattern](#) or [CurrencyPositivePattern](#) property, which returns an integer that determines the following:
 - The placement of the currency symbol.
 - Whether negative values are indicated by a leading negative sign, a trailing negative sign, or parentheses.
 - Whether a space appears between the numeric value and the currency symbol.
- The [CurrencyDecimalDigits](#) property, which defines the number of fractional digits in the result string.
- The [CurrencyDecimalSeparator](#) property, which defines the decimal separator symbol in the result string.
- The [CurrencyGroupSeparator](#) property, which defines the group separator symbol.
- The [CurrencyGroupSizes](#) property, which defines the number of digits in each group to the left of the decimal.
- The [NegativeSign](#) property, which determines the negative sign used in the result string if parentheses are not used to indicate negative values.

In addition, numeric format strings may include a precision specifier. The meaning of this specifier depends on the format string with which it is used, but it typically indicates either the total number of digits or the number of fractional digits that should appear in the result string. For example, the following example uses the "X4" standard numeric string and a precision specifier to create a string value that has four hexadecimal digits.

C#

```
byte[] byteValues = { 12, 163, 255 };
foreach (byte byteValue in byteValues)
    Console.WriteLine(byteValue.ToString("X4"));
// The example displays the following output:
//      000C
//      00A3
//      00FF
```

For more information about standard numeric formatting strings, see [Standard Numeric Format Strings](#).

Standard format strings for date and time values are aliases for custom format strings stored by a particular [DateTimeFormatInfo](#) property. For example, calling the `ToString` method of a date and time value with the "D" format specifier displays the date and

time by using the custom format string stored in the current culture's `DateFormatInfo.LongDatePattern` property. (For more information about custom format strings, see the [next section](#).) The following example illustrates this relationship.

C#

```
using System;
using System.Globalization;

public class Example
{
    public static void Main()
    {
        DateTime date1 = new DateTime(2009, 6, 30);
        Console.WriteLine("D Format Specifier: {0:D}", date1);
        string longPattern =
CultureInfo.CurrentCulture.DateTimeFormat.LongDatePattern;
        Console.WriteLine("{0}' custom format string: {1}",
                           longPattern, date1.ToString(longPattern));
    }
}
// The example displays the following output when run on a system whose
// current culture is en-US:
//      D Format Specifier: Tuesday, June 30, 2009
//      'ddd, MMMM dd, yyyy' custom format string: Tuesday, June 30, 2009
```

For more information about standard date and time format strings, see [Standard Date and Time Format Strings](#).

You can also use standard format strings to define the string representation of an application-defined object that is produced by the object's `ToString(String)` method.

You can define the specific standard format specifiers that your object supports, and you can determine whether they are case-sensitive or case-insensitive. Your implementation of the `ToString(String)` method should support the following:

- A "G" format specifier that represents a customary or common format of the object. The parameterless overload of your object's `ToString` method should call its `ToString(String)` overload and pass it the "G" standard format string.
- Support for a format specifier that is equal to a null reference (`Nothing` in Visual Basic). A format specifier that is equal to a null reference should be considered equivalent to the "G" format specifier.

For example, a `Temperature` class can internally store the temperature in degrees Celsius and use format specifiers to represent the value of the `Temperature` object in degrees Celsius, degrees Fahrenheit, and kelvins. The following example provides an illustration.

C#

```
using System;

public class Temperature
{
    private decimal m_Temp;

    public Temperature(decimal temperature)
    {
        this.m_Temp = temperature;
    }

    public decimal Celsius
    {
        get { return this.m_Temp; }
    }

    public decimal Kelvin
    {
        get { return this.m_Temp + 273.15m; }
    }

    public decimal Fahrenheit
    {
        get { return Math.Round(((decimal) (this.m_Temp * 9 / 5 + 32)), 2); }
    }

    public override string ToString()
    {
        return this.ToString("C");
    }

    public string ToString(string format)
    {
        // Handle null or empty string.
        if (String.IsNullOrEmpty(format)) format = "C";
        // Remove spaces and convert to uppercase.
        format = format.Trim().ToUpperInvariant();

        // Convert temperature to Fahrenheit and return string.
        switch (format)
        {
            // Convert temperature to Fahrenheit and return string.
            case "F":
                return this.Fahrenheit.ToString("N2") + " °F";
            // Convert temperature to Kelvin and return string.
            case "K":
                return this.Kelvin.ToString("N2") + " K";
            // return temperature in Celsius.
            case "G":
            case "C":
                return this.Celsius.ToString("N2") + " °C";
            default:
```

```

        throw new FormatException(String.Format("The '{0}' format string
is not supported.", format));
    }
}
}

public class Example1
{
    public static void Main()
    {
        Temperature temp1 = new Temperature(0m);
        Console.WriteLine(temp1.ToString());
        Console.WriteLine(temp1.ToString("G"));
        Console.WriteLine(temp1.ToString("C"));
        Console.WriteLine(temp1.ToString("F"));
        Console.WriteLine(temp1.ToString("K"));

        Temperature temp2 = new Temperature(-40m);
        Console.WriteLine(temp2.ToString());
        Console.WriteLine(temp2.ToString("G"));
        Console.WriteLine(temp2.ToString("C"));
        Console.WriteLine(temp2.ToString("F"));
        Console.WriteLine(temp2.ToString("K"));

        Temperature temp3 = new Temperature(16m);
        Console.WriteLine(temp3.ToString());
        Console.WriteLine(temp3.ToString("G"));
        Console.WriteLine(temp3.ToString("C"));
        Console.WriteLine(temp3.ToString("F"));
        Console.WriteLine(temp3.ToString("K"));

        Console.WriteLine(String.Format("The temperature is now {0:F}.", 
temp3));
    }
}

// The example displays the following output:
//      0.00 °C
//      0.00 °C
//      0.00 °C
//      32.00 °F
//      273.15 K
//      -40.00 °C
//      -40.00 °C
//      -40.00 °F
//      233.15 K
//      16.00 °C
//      16.00 °C
//      16.00 °C
//      60.80 °F
//      289.15 K
//      The temperature is now 16.00 °C.

```

Custom format strings

In addition to the standard format strings, .NET defines custom format strings for both numeric values and date and time values. A custom format string consists of one or more custom format specifiers that define the string representation of a value. For example, the custom date and time format string "yyyy/mm/dd hh:mm:ss.ffff t zzz" converts a date to its string representation in the form "2008/11/15 07:45:00.0000 P -08:00" for the en-US culture. Similarly, the custom format string "0000" converts the integer value 12 to "0012". For a complete list of custom format strings, see [Custom Date and Time Format Strings](#) and [Custom Numeric Format Strings](#).

If a format string consists of a single custom format specifier, the format specifier should be preceded by the percent (%) symbol to avoid confusion with a standard format specifier. The following example uses the "M" custom format specifier to display a one-digit or two-digit number of the month of a particular date.

C#

```
DateTime date1 = new DateTime(2009, 9, 8);
Console.WriteLine(date1.ToString("%M"));           // Displays 9
```

Many standard format strings for date and time values are aliases for custom format strings that are defined by properties of the [DateTimeFormatInfo](#) object. Custom format strings also offer considerable flexibility in providing application-defined formatting for numeric values or date and time values. You can define your own custom result strings for both numeric values and date and time values by combining multiple custom format specifiers into a single custom format string. The following example defines a custom format string that displays the day of the week in parentheses after the month name, day, and year.

C#

```
string customFormat = "MMMM dd, yyyy (dddd)";
DateTime date1 = new DateTime(2009, 8, 28);
Console.WriteLine(date1.ToString(customFormat));
// The example displays the following output if run on a system
// whose language is English:
//     August 28, 2009 (Friday)
```

The following example defines a custom format string that displays an [Int64](#) value as a standard, seven-digit U.S. telephone number along with its area code.

C#

```

using System;

public class Example17
{
    public static void Main()
    {
        long number = 800999999;
        string fmt = "000-000-0000";
        Console.WriteLine(number.ToString(fmt));
    }
}
// The example displays the following output:
//      800-999-9999

```

Although standard format strings can generally handle most of the formatting needs for your application-defined types, you may also define custom format specifiers to format your types.

Format strings and .NET types

All numeric types (that is, the [Byte](#), [Decimal](#), [Double](#), [Int16](#), [Int32](#), [Int64](#), [SByte](#), [Single](#), [UInt16](#), [UInt32](#), [UInt64](#), and [BigInteger](#) types), as well as the [DateTime](#), [DateTimeOffset](#), [TimeSpan](#), [Guid](#), and all enumeration types, support formatting with format strings. For information on the specific format strings supported by each type, see the following topics:

| Title | Definition |
|---|---|
| Standard Numeric Format Strings | Describes standard format strings that create commonly used string representations of numeric values. |
| Custom Numeric Format Strings | Describes custom format strings that create application-specific formats for numeric values. |
| Standard Date and Time Format Strings | Describes standard format strings that create commonly used string representations of DateTime and DateTimeOffset values. |
| Custom Date and Time Format Strings | Describes custom format strings that create application-specific formats for DateTime and DateTimeOffset values. |
| Standard TimeSpan Format Strings | Describes standard format strings that create commonly used string representations of time intervals. |
| Custom TimeSpan Format Strings | Describes custom format strings that create application-specific formats for time intervals. |
| Enumeration Format Strings | Describes standard format strings that are used to create string |

| Title | Definition |
|-----------------------|--|
| | representations of enumeration values. |
| Guid.ToString(String) | Describes standard format strings for Guid values. |

Culture-sensitive formatting with format providers

Although format specifiers let you customize the formatting of objects, producing a meaningful string representation of objects often requires additional formatting information. For example, formatting a number as a currency value by using either the "C" standard format string or a custom format string such as "\$ #,#.00" requires, at a minimum, information about the correct currency symbol, group separator, and decimal separator to be available to include in the formatted string. In .NET, this additional formatting information is made available through the [IFormatProvider](#) interface, which is provided as a parameter to one or more overloads of the `ToString` method of numeric types and date and time types. [IFormatProvider](#) implementations are used in .NET to support culture-specific formatting. The following example illustrates how the string representation of an object changes when it is formatted with three [IFormatProvider](#) objects that represent different cultures.

C#

```
using System;
using System.Globalization;

public class Example18
{
    public static void Main()
    {
        decimal value = 1603.42m;
        Console.WriteLine(value.ToString("C3", new CultureInfo("en-US")));
        Console.WriteLine(value.ToString("C3", new CultureInfo("fr-FR")));
        Console.WriteLine(value.ToString("C3", new CultureInfo("de-DE")));
    }
}
// The example displays the following output:
//      $1,603.420
//      1 603,420 €
//      1.603,420 €
```

The [IFormatProvider](#) interface includes one method, [GetFormat\(Type\)](#), which has a single parameter that specifies the type of object that provides formatting information. If the

method can provide an object of that type, it returns it. Otherwise, it returns a null reference (`Nothing` in Visual Basic).

[IFormatProvider.GetFormat](#) is a callback method. When you call a `ToString` method overload that includes an [IFormatProvider](#) parameter, it calls the [GetFormat](#) method of that [IFormatProvider](#) object. The [GetFormat](#) method is responsible for returning an object that provides the necessary formatting information, as specified by its `formatType` parameter, to the `ToString` method.

A number of formatting or string conversion methods include a parameter of type [IFormatProvider](#), but in many cases the value of the parameter is ignored when the method is called. The following table lists some of the formatting methods that use the parameter and the type of the [Type](#) object that they pass to the [IFormatProvider.GetFormat](#) method.

| Method | Type of <code>formatType</code> parameter |
|---|--|
| <code>ToString</code> method of numeric types | <code>System.Globalization.NumberFormatInfo</code> |
| <code>ToString</code> method of date and time types | <code>System.Globalization.DateTimeFormatInfo</code> |
| <code>String.Format</code> | <code>System.ICustomFormatter</code> |
| <code>StringBuilder.AppendFormat</code> | <code>System.ICustomFormatter</code> |

① Note

The `ToString` methods of the numeric types and date and time types are overloaded, and only some of the overloads include an [IFormatProvider](#) parameter. If a method does not have a parameter of type [IFormatProvider](#), the object that is returned by the [CultureInfo.CurrentCulture](#) property is passed instead. For example, a call to the default [Int32.ToString\(\)](#) method ultimately results in a method call such as the following: `Int32.ToString("G", System.Globalization.CultureInfo.CurrentCulture)`.

.NET provides three classes that implement [IFormatProvider](#):

- [DateTimeFormatInfo](#), a class that provides formatting information for date and time values for a specific culture. Its [IFormatProvider.GetFormat](#) implementation returns an instance of itself.
- [NumberFormatInfo](#), a class that provides numeric formatting information for a specific culture. Its [IFormatProvider.GetFormat](#) implementation returns an instance

of itself.

- **CultureInfo**. Its [IFormatProvider.GetFormat](#) implementation can return either a **NumberFormatInfo** object to provide numeric formatting information or a **DateTimeFormatInfo** object to provide formatting information for date and time values.

You can also implement your own format provider to replace any one of these classes. However, your implementation's [GetFormat](#) method must return an object of the type listed in the previous table if it has to provide formatting information to the [ToString](#) method.

Culture-sensitive formatting of numeric values

By default, the formatting of numeric values is culture-sensitive. If you do not specify a culture when you call a formatting method, the formatting conventions of the current culture are used. This is illustrated in the following example, which changes the current culture four times and then calls the [Decimal.ToString\(String\)](#) method. In each case, the result string reflects the formatting conventions of the current culture. This is because the [ToString](#) and [ToString\(String\)](#) methods wrap calls to each numeric type's [ToString\(String, IFormatProvider\)](#) method.

C#

```
using System.Globalization;

public class Example6
{
    public static void Main()
    {
        string[] cultureNames = { "en-US", "fr-FR", "es-MX", "de-DE" };
        Decimal value = 1043.17m;

        foreach (var cultureName in cultureNames) {
            // Change the current culture.
            CultureInfo.CurrentCulture =
            CultureInfo.CreateSpecificCulture(cultureName);
            Console.WriteLine($"The current culture is
{CultureInfo.CurrentCulture.Name}");
            Console.WriteLine(value.ToString("C2"));
            Console.WriteLine();
        }
    }
} // The example displays the following output:
//      The current culture is en-US
//      $1,043.17
//
```

```
//      The current culture is fr-FR
//      1 043,17 €
//
//      The current culture is es-MX
//      $1,043.17
//
//      The current culture is de-DE
//      1.043,17 €
```

You can also format a numeric value for a specific culture by calling a `ToString` overload that has a `provider` parameter and passing it either of the following:

- A `CultureInfo` object that represents the culture whose formatting conventions are to be used. Its `CultureInfo.GetFormat` method returns the value of the `CultureInfo.NumberFormat` property, which is the `NumberFormatInfo` object that provides culture-specific formatting information for numeric values.
- A `NumberFormatInfo` object that defines the culture-specific formatting conventions to be used. Its `GetFormat` method returns an instance of itself.

The following example uses `NumberFormatInfo` objects that represent the English (United States) and English (United Kingdom) cultures and the French and Russian neutral cultures to format a floating-point number.

C#

```
using System.Globalization;

public class Example7
{
    public static void Main()
    {
        double value = 1043.62957;
        string[] cultureNames = { "en-US", "en-GB", "ru", "fr" };

        foreach (string? name in cultureNames)
        {
            NumberFormatInfo nfi =
CultureInfo.CreateSpecificCulture(name).NumberFormat;
            Console.WriteLine("{0,-6} {1}", name + ":", value.ToString("N3",
nfi));
        }
    }
}
// The example displays the following output:
//      en-US: 1,043.630
//      en-GB: 1,043.630
//      ru:      1 043,630
//      fr:      1 043,630
```

Culture-sensitive formatting of date and time values

By default, the formatting of date and time values is culture-sensitive. If you do not specify a culture when you call a formatting method, the formatting conventions of the current culture are used. This is illustrated in the following example, which changes the current culture four times and then calls the [DateTime.ToString\(String\)](#) method. In each case, the result string reflects the formatting conventions of the current culture. This is because the [DateTime.ToString\(\)](#), [DateTime.ToString\(String\)](#), [DateTimeOffset.ToString\(\)](#), and [DateTimeOffset.ToString\(String\)](#) methods wrap calls to the [DateTime.ToString\(String, IFormatProvider\)](#) and [DateTimeOffset.ToString\(String, IFormatProvider\)](#) methods.

C#

```
using System.Globalization;

public class Example4
{
    public static void Main()
    {
        string[] cultureNames = { "en-US", "fr-FR", "es-MX", "de-DE" };
        DateTime dateToFormat = new DateTime(2012, 5, 28, 11, 30, 0);

        foreach (var cultureName in cultureNames) {
            // Change the current culture.
            CultureInfo.CurrentCulture =
            CultureInfo.CreateSpecificCulture(cultureName);
            Console.WriteLine($"The current culture is
{CultureInfo.CurrentCulture.Name}");
            Console.WriteLine(dateToFormat.ToString("F"));
            Console.WriteLine();
        }
    }
}

// The example displays the following output:
//      The current culture is en-US
//      Monday, May 28, 2012 11:30:00 AM
//
//      The current culture is fr-FR
//      lundi 28 mai 2012 11:30:00
//
//      The current culture is es-MX
//      lunes, 28 de mayo de 2012 11:30:00 a.m.
//
//      The current culture is de-DE
//      Montag, 28. Mai 2012 11:30:00
```

You can also format a date and time value for a specific culture by calling a [DateTime.ToString](#) or [DateTimeOffset.ToString](#) overload that has a `provider` parameter and passing it either of the following:

- A [CultureInfo](#) object that represents the culture whose formatting conventions are to be used. Its [CultureInfo.GetFormat](#) method returns the value of the [CultureInfo.DateTimeFormat](#) property, which is the [DateTimeFormatInfo](#) object that provides culture-specific formatting information for date and time values.
- A [DateTimeFormatInfo](#) object that defines the culture-specific formatting conventions to be used. Its [GetFormat](#) method returns an instance of itself.

The following example uses [DateTimeFormatInfo](#) objects that represent the English (United States) and English (United Kingdom) cultures and the French and Russian neutral cultures to format a date.

C#

```
using System.Globalization;

public class Example5
{
    public static void Main()
    {
        DateTime dat1 = new(2012, 5, 28, 11, 30, 0);
        string[] cultureNames = { "en-US", "en-GB", "ru", "fr" };

        foreach (var name in cultureNames) {
            DateTimeFormatInfo dtfi =
                CultureInfo.CreateSpecificCulture(name).DateTimeFormat;
            Console.WriteLine($"{name}: {dat1.ToString(dtfi)}");
        }
    }
}
// The example displays the following output:
//      en-US: 5/28/2012 11:30:00 AM
//      en-GB: 28/05/2012 11:30:00
//      ru: 28.05.2012 11:30:00
//      fr: 28/05/2012 11:30:00
```

The [IFormattable](#) interface

Typically, types that overload the [ToString](#) method with a format string and an [IFormatProvider](#) parameter also implement the [IFormattable](#) interface. This interface has a single member, [IFormattable.ToString\(String, IFormatProvider\)](#), that includes both a format string and a format provider as parameters.

Implementing the [IFormattable](#) interface for your application-defined class offers two advantages:

- Support for string conversion by the [Convert](#) class. Calls to the [Convert.ToString\(Object\)](#) and [Convert.ToString\(Object, IFormatProvider\)](#) methods call your [IFormattable](#) implementation automatically.
- Support for composite formatting. If a format item that includes a format string is used to format your custom type, the common language runtime automatically calls your [IFormattable](#) implementation and passes it the format string. For more information about composite formatting with methods such as [String.Format](#) or [Console.WriteLine](#), see the [Composite Formatting](#) section.

The following example defines a `Temperature` class that implements the [IFormattable](#) interface. It supports the "C" or "G" format specifiers to display the temperature in Celsius, the "F" format specifier to display the temperature in Fahrenheit, and the "K" format specifier to display the temperature in Kelvin.

C#

```
using System;
using System.Globalization;

namespace HotAndCold
{
    public class Temperature : IFormattable
    {
        private decimal m_Temp;

        public Temperature(decimal temperature)
        {
            this.m_Temp = temperature;
        }

        public decimal Celsius
        {
            get { return this.m_Temp; }
        }

        public decimal Kelvin
        {
            get { return this.m_Temp + 273.15m; }
        }

        public decimal Fahrenheit
        {
            get { return Math.Round((decimal)this.m_Temp * 9 / 5 + 32, 2); }
        }

        public override string ToString()
        {
            return this.ToString("G", null);
        }
    }
}
```

```

    }

    public string ToString(string format)
    {
        return this.ToString(format, null);
    }

    public string ToString(string format, IFormatProvider provider)
    {
        // Handle null or empty arguments.
        if (String.IsNullOrEmpty(format))
            format = "G";
        // Remove any white space and convert to uppercase.
        format = format.Trim().ToUpperInvariant();

        if (provider == null)
            provider = NumberFormatInfo.CurrentInfo;

        switch (format)
        {
            // Convert temperature to Fahrenheit and return string.
            case "F":
                return this.Fahrenheit.ToString("N2", provider) + °F";
            // Convert temperature to Kelvin and return string.
            case "K":
                return this.Kelvin.ToString("N2", provider) + "K";
            // Return temperature in Celsius.
            case "C":
            case "G":
                return this.Celsius.ToString("N2", provider) + °C";
            default:
                throw new FormatException(String.Format("The '{0}'"
format string is not supported.", format));
        }
    }
}

```

The following example instantiates a `Temperature` object. It then calls the `ToString` method and uses several composite format strings to obtain different string representations of a `Temperature` object. Each of these method calls, in turn, calls the `IFormattable` implementation of the `Temperature` class.

C#

```

public class Example11
{
    public static void Main()
    {
        CultureInfo.CurrentCulture = CultureInfo.GetCultureInfo("en-US");
        Temperature temp = new Temperature(22m);
        Console.WriteLine(Convert.ToString(temp, new CultureInfo("ja-JP")));
        Console.WriteLine("Temperature: {0}K", temp);
    }
}

```

```

        Console.WriteLine("Temperature: {0:F}", temp);
        Console.WriteLine(String.Format(new CultureInfo("fr-FR"),
"Temperature: {0:F}", temp));
    }
}
// The example displays the following output:
//      22.00°C
//      Temperature: 295.15K
//      Temperature: 71.60°F
//      Temperature: 71,60°F

```

Composite formatting

Some methods, such as [String.Format](#) and [StringBuilder.AppendFormat](#), support composite formatting. A composite format string is a kind of template that returns a single string that incorporates the string representation of zero, one, or more objects. Each object is represented in the composite format string by an indexed format item. The index of the format item corresponds to the position of the object that it represents in the method's parameter list. Indexes are zero-based. For example, in the following call to the [String.Format](#) method, the first format item, `{0:D}`, is replaced by the string representation of `thatDate`; the second format item, `{1}`, is replaced by the string representation of `item1`; and the third format item, `{2:C2}`, is replaced by the string representation of `item1.Value`.

C#

```

result = String.Format("On {0:d}, the inventory of {1} was worth {2:C2}.",
                      thatDate, item1, item1.Value);
Console.WriteLine(result);
// The example displays output like the following if run on a system
// whose current culture is en-US:
//      On 5/1/2009, the inventory of WidgetA was worth $107.44.

```

In addition to replacing a format item with the string representation of its corresponding object, format items also let you control the following:

- The specific way in which an object is represented as a string, if the object implements the [IFormattable](#) interface and supports format strings. You do this by following the format item's index with a `:` (colon) followed by a valid format string. The previous example did this by formatting a date value with the "d" (short date pattern) format string (for example, `{0:d}`) and by formatting a numeric value with the "C2" format string (for example, `{2:c2}` to represent the number as a currency value with two fractional decimal digits.

- The width of the field that contains the object's string representation, and the alignment of the string representation in that field. You do this by following the format item's index with a `,` (comma) followed the field width. The string is right-aligned in the field if the field width is a positive value, and it is left-aligned if the field width is a negative value. The following example left-aligns date values in a 20-character field, and it right-aligns decimal values with one fractional digit in an 11-character field.

C#

```
DateTime startDate = new DateTime(2015, 8, 28, 6, 0, 0);
decimal[] temps = { 73.452m, 68.98m, 72.6m, 69.24563m,
                    74.1m, 72.156m, 72.228m };
Console.WriteLine("{0,-20} {1,11}\n", "Date", "Temperature");
for (int ctr = 0; ctr < temps.Length; ctr++)
    Console.WriteLine("{0,-20:g} {1,11:N1}", startDate.AddDays(ctr),
                      temps[ctr]);

// The example displays the following output:
//          Date                  Temperature
//
//        8/28/2015 6:00 AM      73.5
//        8/29/2015 6:00 AM      69.0
//        8/30/2015 6:00 AM      72.6
//        8/31/2015 6:00 AM      69.2
//        9/1/2015 6:00 AM       74.1
//        9/2/2015 6:00 AM       72.2
//        9/3/2015 6:00 AM       72.2
```

Note that, if both the alignment string component and the format string component are present, the former precedes the latter (for example, `{0,-20:g}`).

For more information about composite formatting, see [Composite Formatting](#).

Custom formatting with `ICustomFormatter`

Two composite formatting methods, `String.Format(IFormatProvider, String, Object[])` and `StringBuilder.AppendFormat(IFormatProvider, String, Object[])`, include a format provider parameter that supports custom formatting. When either of these formatting methods is called, it passes a `Type` object that represents an `ICustomFormatter` interface to the format provider's `GetFormat` method. The `GetFormat` method is then responsible for returning the `ICustomFormatter` implementation that provides custom formatting.

The `ICustomFormatter` interface has a single method, `Format(String, Object, IFormatProvider)`, that is called automatically by a composite formatting method, once for each format item in a composite format string. The `Format(String, Object,`

`IFormatProvider`) method has three parameters: a format string, which represents the `formatString` argument in a format item, an object to format, and an `IFormatProvider` object that provides formatting services. Typically, the class that implements `ICustomFormatter` also implements `IFormatProvider`, so this last parameter is a reference to the custom formatting class itself. The method returns a custom formatted string representation of the object to be formatted. If the method cannot format the object, it should return a null reference (`Nothing` in Visual Basic).

The following example provides an `ICustomFormatter` implementation named `ByteByByteFormatter` that displays integer values as a sequence of two-digit hexadecimal values followed by a space.

C#

```
public class ByteByByteFormatter : IFormatProvider, ICustomFormatter
{
    public object GetFormat(Type formatType)
    {
        if (formatType == typeof(ICustomFormatter))
            return this;
        else
            return null;
    }

    public string Format(string format, object arg,
                         IFormatProvider formatProvider)
    {
        if (!formatProvider.Equals(this)) return null;

        // Handle only hexadecimal format string.
        if (!format.StartsWith("X")) return null;

        byte[] bytes;
        string output = null;

        // Handle only integral types.
        if (arg is Byte)
            bytes = BitConverter.GetBytes((Byte) arg);
        else if (arg is Int16)
            bytes = BitConverter.GetBytes((Int16) arg);
        else if (arg is Int32)
            bytes = BitConverter.GetBytes((Int32) arg);
        else if (arg is Int64)
            bytes = BitConverter.GetBytes((Int64) arg);
        else if (arg is SByte)
            bytes = BitConverter.GetBytes((SByte) arg);
        else if (arg is UInt16)
            bytes = BitConverter.GetBytes((UInt16) arg);
        else if (arg is UInt32)
            bytes = BitConverter.GetBytes((UInt32) arg);
        else if (arg is UInt64)
```

```

        bytes = BitConverter.GetBytes((UInt64) arg);
    else
        return null;

    for (int ctr = bytes.Length - 1; ctr >= 0; ctr--)
        output += String.Format("{0:X2} ", bytes[ctr]);

    return output.Trim();
}
}

```

The following example uses the `ByteByByteFormatter` class to format integer values. Note that the `ICustomFormatter.Format` method is called more than once in the second `String.Format(IFormatProvider, String, Object[])` method call, and that the default `NumberFormatInfo` provider is used in the third method call because the `.ByteByByteFormatter.Format` method does not recognize the "N0" format string and returns a null reference (`Nothing` in Visual Basic).

C#

```

public class Example10
{
    public static void Main()
    {
        long value = 3210662321;
        byte value1 = 214;
        byte value2 = 19;

        Console.WriteLine(String.Format(new ByteByByteFormatter(), "{0:X}",
value));
        Console.WriteLine(String.Format(new ByteByByteFormatter(), "{0:X} And
{1:X} = {2:X} ({2:000})", value1, value2, value1 & value2));
        Console.WriteLine(String.Format(new ByteByByteFormatter(), "
{0,10:N0}", value));
    }
}

// The example displays the following output:
//      00 00 00 00 BF 5E D1 B1
//      00 D6 And 00 13 = 00 12 (018)
//      3,210,662,321

```

See also

| Title | Definition |
|---|---|
| Standard Numeric Format Strings | Describes standard format strings that create commonly used string representations of numeric values. |

| Title | Definition |
|---|---|
| Custom Numeric Format Strings | Describes custom format strings that create application-specific formats for numeric values. |
| Standard Date and Time Format Strings | Describes standard format strings that create commonly used string representations of DateTime values. |
| Custom Date and Time Format Strings | Describes custom format strings that create application-specific formats for DateTime values. |
| Standard TimeSpan Format Strings | Describes standard format strings that create commonly used string representations of time intervals. |
| Custom TimeSpan Format Strings | Describes custom format strings that create application-specific formats for time intervals. |
| Enumeration Format Strings | Describes standard format strings that are used to create string representations of enumeration values. |
| Composite Formatting | Describes how to embed one or more formatted values in a string. The string can subsequently be displayed on the console or written to a stream. |
| Parsing Strings | Describes how to initialize objects to the values described by string representations of those objects. Parsing is the inverse operation of formatting. |

Reference

- [System.IFormattable](#)
- [System.IFormatProvider](#)
- [System.ILazyFormatProvider](#)

 Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback [here](#).

 [Open a documentation issue](#)

 [Provide product feedback](#)

Standard numeric format strings

Article • 04/25/2023

Standard numeric format strings are used to format common numeric types. A standard numeric format string takes the form `[format specifier][precision specifier]`, where:

- *Format specifier* is a single alphabetic character that specifies the type of number format, for example, currency or percent. Any numeric format string that contains more than one alphabetic character, including white space, is interpreted as a custom numeric format string. For more information, see [Custom numeric format strings](#).
- *Precision specifier* is an optional integer that affects the number of digits in the resulting string. In .NET 7 and later versions, the maximum precision value is 999,999,999. In .NET 6, the maximum precision value is `Int32.MaxValue`. In previous .NET versions, the precision can range from 0 to 99. The precision specifier controls the number of digits in the string representation of a number. It does not round the number itself. To perform a rounding operation, use the `Math.Ceiling`, `Math.Floor`, or `Math.Round` method.

When *precision specifier* controls the number of fractional digits in the result string, the result string reflects a number that is rounded to a representable result nearest to the infinitely precise result. If there are two equally near representable results:

- On .NET Framework and .NET Core up to .NET Core 2.0, the runtime selects the result with the greater least significant digit (that is, using `MidpointRounding.AwayFromZero`).
- On .NET Core 2.1 and later, the runtime selects the result with an even least significant digit (that is, using `MidpointRounding.ToEven`).

(!) Note

The precision specifier determines the number of digits in the result string. To pad a result string with leading or trailing spaces, use the [composite formatting](#) feature and define an *alignment component* in the format item.

Standard numeric format strings are supported by:

- Some overloads of the `ToString` method of all numeric types. For example, you can supply a numeric format string to the `Int32.ToString(String)` and `Int32.ToString(String, IFormatProvider)` methods.

- The `TryFormat` method of all numeric types, for example, `Int32.TryParse(Span<Char>, Int32, ReadOnlySpan<Char>, IFormatProvider)` and `Single.TryParse(Span<Char>, Int32, ReadOnlySpan<Char>, IFormatProvider)`.
- The .NET [composite formatting feature](#), which is used by some `Write` and `WriteLine` methods of the `Console` and `StreamWriter` classes, the `String.Format` method, and the `StringBuilder.AppendFormat` method. The composite format feature allows you to include the string representation of multiple data items in a single string, to specify field width, and to align numbers in a field. For more information, see [Composite Formatting](#).
- [Interpolated strings](#) in C# and Visual Basic, which provide a simplified syntax when compared to composite format strings.

Tip

You can download the [Formatting Utility](#), a .NET Core Windows Forms application that lets you apply format strings to either numeric or date and time values and displays the result string. Source code is available for [C#](#) and [Visual Basic](#).

Standard format specifiers

The following table describes the standard numeric format specifiers and displays sample output produced by each format specifier. See the [Notes](#) section for additional information about using standard numeric format strings, and the [Code example](#) section for a comprehensive illustration of their use.

The result of a formatted string for a specific culture might differ from the following examples. Operating system settings, user settings, environment variables, and the .NET version you're using can all affect the format. For example, starting with .NET 5, .NET tries to unify cultural formats across platforms. For more information, see [.NET globalization and ICU](#).

| Format specifier | Name | Description | Examples |
|------------------|--------|--|--|
| "B" or "b" | Binary | <p>Result: A binary string.</p> <p>Supported by: Integral types only (.NET 8+).</p> <p>Precision specifier: Number of digits in the result string.</p> | <p>42 ("B") -> 101010</p> <p>255 ("b16") -> 0000000011111111</p> |

| Format specifier | Name | Description | Examples |
|-------------------------|--------------------------|---|--|
| | | More information: The Binary ("B") Format Specifier . | |
| "C" or "c" | Currency | <p>Result: A currency value.</p> <p>Supported by: All numeric types.</p> <p>Precision specifier: Number of decimal digits.</p> <p>Default precision specifier: Defined by NumberFormatInfo.CurrencyDecimalDigits.</p> <p>More information: The Currency ("C") Format Specifier.</p> | 123.456 ("C", en-US) -> \$123.46 123.456 ("C", fr-FR) -> 123,46 € 123.456 ("C", ja-JP) -> ¥123 -123.456 ("C3", en-US) -> (\$123.456) -123.456 ("C3", fr-FR) -> -123,456 € -123.456 ("C3", ja-JP) -> -¥123.456 |
| "D" or "d" | Decimal | <p>Result: Integer digits with optional negative sign.</p> <p>Supported by: Integral types only.</p> <p>Precision specifier: Minimum number of digits.</p> <p>Default precision specifier: Minimum number of digits required.</p> <p>More information: The Decimal("D") Format Specifier.</p> | 1234 ("D") -> 1234 -1234 ("D6") -> -001234 |
| "E" or "e" | Exponential (scientific) | <p>Result: Exponential notation.</p> <p>Supported by: All numeric types.</p> <p>Precision specifier: Number of decimal digits.</p> <p>Default precision specifier: 6.</p> <p>More information: The Exponential ("E") Format Specifier.</p> | 1052.0329112756 ("E", en-US) -> 1.052033E+003 1052.0329112756 ("e", fr-FR) -> 1,052033e+003 -1052.0329112756 ("e2", en-US) -> -1.05e+003 |

| Format specifier | Name | Description | Examples |
|-------------------------|-------------|---|--|
| | | | -1052.0329112756 ("E2", fr-FR) -> -1,05E+003 |
| "F" or "f" | Fixed-point | <p>Result: Integral and decimal digits with optional negative sign.</p> <p>Supported by: All numeric types.</p> <p>Precision specifier: Number of decimal digits.</p> <p>Default precision specifier: Defined by NumberFormatInfo.NumberDecimalDigits.</p> <p>More information: The Fixed-Point ("F") Format Specifier.</p> | 1234.567 ("F", en-US) -> 1234.57 1234.567 ("F", de-DE) -> 1234,57 1234 ("F1", en-US) -> 1234.0 1234 ("F1", de-DE) -> 1234,0 -1234.56 ("F4", en-US) -> -1234.5600 -1234.56 ("F4", de-DE) -> -1234,5600 |
| "G" or "g" | General | <p>Result: The more compact of either fixed-point or scientific notation.</p> <p>Supported by: All numeric types.</p> <p>Precision specifier: Number of significant digits.</p> <p>Default precision specifier: Depends on numeric type.</p> <p>More information: The General ("G") Format Specifier.</p> | -123.456 ("G", en-US) -> -123.456 -123.456 ("G", sv-SE) -> -123,456 123.4546 ("G4", en-US) -> 123.5 123.4546 ("G4", sv-SE) -> 123,5 -1.234567890e-25 ("G", en-US) -> -1.23456789E-25 -1.234567890e-25 ("G", sv-SE) -> -1,23456789E-25 |
| "N" or "n" | Number | Result: Integral and decimal digits, group separators, and a decimal separator with optional negative sign. | 1234.567 ("N", en-US) -> 1,234.57 1234.567 ("N", ru-RU) |

| Format specifier | Name | Description | Examples |
|------------------|-------------|---|---|
| | | Supported by: All numeric types. | -> 1 234,57 |
| | | Precision specifier: Desired number of decimal places. | 1234 ("N1", en-US) -> 1,234.0 |
| | | Default precision specifier: Defined by NumberFormatInfo.NumberDecimalDigits . | 1234 ("N1", ru-RU) -> 1 234,0 |
| | | More information: The Numeric ("N") Format Specifier . | -1234.56 ("N3", en-US) -> -1,234.560 |
| | | | -1234.56 ("N3", ru-RU) -> -1 234,560 |
| "P" or "p" | Percent | Result: Number multiplied by 100 and displayed with a percent symbol. | 1 ("P", en-US) -> 100.00 % |
| | | Supported by: All numeric types. | 1 ("P", fr-FR) -> 100,00 % |
| | | Precision specifier: Desired number of decimal places. | -0.39678 ("P1", en-US) -> -39.7 % |
| | | Default precision specifier: Defined by NumberFormatInfo.PercentDecimalDigits . | -0.39678 ("P1", fr-FR) -> -39,7 % |
| | | More information: The Percent ("P") Format Specifier . | |
| "R" or "r" | Round-trip | Result: A string that can round-trip to an identical number. Supported by: Single , Double , and BigInteger . Note: Recommended for the BigInteger type only. For Double types, use "G17"; for Single types, use "G9". Precision specifier: Ignored. | 123456789.12345678 ("R") -> 123456789.12345678 -1234567890.12345678 ("R") -> -1234567890.1234567 |
| | | More information: The Round-trip ("R") Format Specifier . | |
| "X" or "x" | Hexadecimal | Result: A hexadecimal string. Supported by: Integral types only. Precision specifier: Number of digits in the | 255 ("X") -> FF -1 ("x") -> ff |

| Format specifier | Name | Description | Examples |
|----------------------------|-------------------|--|---|
| | | <p>result string.</p> <p>More information: The Hexadecimal ("X") Format Specifier.</p> | 255 ("x4") -> 00ff -1 ("X4") -> 00FF |
| Any other single character | Unknown specifier | Result: Throws a FormatException at run time. | |

Use standard numeric format strings

ⓘ Note

The C# examples in this article run in the Try.NET inline code runner and playground. Select the **Run** button to run an example in an interactive window. Once you execute the code, you can modify it and run the modified code by selecting **Run** again. The modified code either runs in the interactive window or, if compilation fails, the interactive window displays all C# compiler error messages.

A standard numeric format string can be used to define the formatting of a numeric value in one of the following ways:

- It can be passed to the `TryFormat` method or an overload of the `ToString` method that has a `format` parameter. The following example formats a numeric value as a currency string in the current culture (in this case, the en-US culture).

C#

```
decimal value = 123.456m;
Console.WriteLine(value.ToString("C2"));
// Displays $123.46
```

- It can be supplied as the `formatString` argument in a format item used with such methods as `String.Format`, `Console.WriteLine`, and `StringBuilder.AppendFormat`. For more information, see [Composite Formatting](#). The following example uses a format item to insert a currency value in a string.

C#

```
decimal value = 123.456m;
Console.WriteLine("Your account balance is {0:C2}.", value);
// Displays "Your account balance is $123.46."
```

Optionally, you can supply an `alignment` argument to specify the width of the numeric field and whether its value is right- or left-aligned. The following example left-aligns a currency value in a 28-character field, and it right-aligns a currency value in a 14-character field.

C#

```
decimal[] amounts = { 16305.32m, 18794.16m };
Console.WriteLine("    Beginning Balance          Ending Balance");
Console.WriteLine("    {0,-28:C2}{1,14:C2}", amounts[0], amounts[1]);
// Displays:
//      Beginning Balance          Ending Balance
//      $16,305.32                $18,794.16
```

- It can be supplied as the `formatString` argument in an interpolated expression item of an interpolated string. For more information, see the [String interpolation](#) article in the C# reference or the [Interpolated strings](#) article in the Visual Basic reference.

The following sections provide detailed information about each of the standard numeric format strings.

Binary format specifier (B)

The binary ("B") format specifier converts a number to a string of binary digits. This format is supported only for integral types and only on .NET 8+.

The precision specifier indicates the minimum number of digits desired in the resulting string. If required, the number is padded with zeros to its left to produce the number of digits given by the precision specifier.

The result string is not affected by the formatting information of the current [NumberFormatInfo](#) object.

Currency format specifier (C)

The "C" (or currency) format specifier converts a number to a string that represents a currency amount. The precision specifier indicates the desired number of decimal places in the result string. If the precision specifier is omitted, the default precision is defined by the [NumberFormatInfo.CurrencyDecimalDigits](#) property.

If the value to be formatted has more than the specified or default number of decimal places, the fractional value is rounded in the result string. If the value to the right of the number of specified decimal places is 5 or greater, the last digit in the result string is rounded away from zero.

The result string is affected by the formatting information of the current [NumberFormatInfo](#) object. The following table lists the [NumberFormatInfo](#) properties that control the formatting of the returned string.

| NumberFormatInfo property | Description |
|--|---|
| CurrencyPositivePattern | Defines the placement of the currency symbol for positive values. |
| CurrencyNegativePattern | Defines the placement of the currency symbol for negative values, and specifies whether the negative sign is represented by parentheses or the NegativeSign property. |
| NegativeSign | Defines the negative sign used if CurrencyNegativePattern indicates that parentheses are not used. |
| CurrencySymbol | Defines the currency symbol. |
| CurrencyDecimalDigits | Defines the default number of decimal digits in a currency value. This value can be overridden by using the precision specifier. |
| CurrencyDecimalSeparator | Defines the string that separates integral and decimal digits. |
| CurrencyGroupSeparator | Defines the string that separates groups of integral numbers. |
| CurrencyGroupSizes | Defines the number of integer digits that appear in a group. |

The following example formats a [Double](#) value with the currency format specifier:

```
C#  
  
double value = 12345.6789;  
Console.WriteLine(value.ToString("C", CultureInfo.CurrentCulture));  
  
Console.WriteLine(value.ToString("C3", CultureInfo.CurrentCulture));  
  
Console.WriteLine(value.ToString("C3",  
        CultureInfo.CreateSpecificCulture("da-DK")));  
// The example displays the following output on a system whose
```

```
// current culture is English (United States):
//      $12,345.68
//      $12,345.679
//      12.345,679 kr
```

Decimal format specifier (D)

The "D" (or decimal) format specifier converts a number to a string of decimal digits (0-9), prefixed by a minus sign if the number is negative. This format is supported only for integral types.

The precision specifier indicates the minimum number of digits desired in the resulting string. If required, the number is padded with zeros to its left to produce the number of digits given by the precision specifier. If no precision specifier is specified, the default is the minimum value required to represent the integer without leading zeros.

The result string is affected by the formatting information of the current [NumberFormatInfo](#) object. As the following table shows, a single property affects the formatting of the result string.

| NumberFormatInfo property | Description |
|------------------------------|--|
| NegativeSign | Defines the string that indicates that a number is negative. |

The following example formats an [Int32](#) value with the decimal format specifier.

```
C#
int value;

value = 12345;
Console.WriteLine(value.ToString("D"));
// Displays 12345
Console.WriteLine(value.ToString("D8"));
// Displays 00012345

value = -12345;
Console.WriteLine(value.ToString("D"));
// Displays -12345
Console.WriteLine(value.ToString("D8"));
// Displays -00012345
```

Exponential format specifier (E)

The exponential ("E") format specifier converts a number to a string of the form "-d.ddd...E+ddd" or "-d.ddd...e+ddd", where each "d" indicates a digit (0-9). The string starts with a minus sign if the number is negative. Exactly one digit always precedes the decimal point.

The precision specifier indicates the desired number of digits after the decimal point. If the precision specifier is omitted, a default of six digits after the decimal point is used.

The case of the format specifier indicates whether to prefix the exponent with an "E" or an "e". The exponent always consists of a plus or minus sign and a minimum of three digits. The exponent is padded with zeros to meet this minimum, if required.

The result string is affected by the formatting information of the current [NumberFormatInfo](#) object. The following table lists the [NumberFormatInfo](#) properties that control the formatting of the returned string.

| NumberFormatInfo property | Description |
|--|--|
| NegativeSign | Defines the string that indicates that a number is negative for both the coefficient and exponent. |
| NumberDecimalSeparator | Defines the string that separates the integral digit from decimal digits in the coefficient. |
| PositiveSign | Defines the string that indicates that an exponent is positive. |

The following example formats a [Double](#) value with the exponential format specifier:

C#

```
double value = 12345.6789;
Console.WriteLine(value.ToString("E", CultureInfo.InvariantCulture));
// Displays 1.234568E+004

Console.WriteLine(value.ToString("E10", CultureInfo.InvariantCulture));
// Displays 1.2345678900E+004

Console.WriteLine(value.ToString("e4", CultureInfo.InvariantCulture));
// Displays 1.2346e+004

Console.WriteLine(value.ToString("E",
    CultureInfo.CreateSpecificCulture("fr-FR")));
// Displays 1,234568E+004
```

Fixed-point format specifier (F)

The fixed-point ("F") format specifier converts a number to a string of the form "-ddd.ddd..." where each "d" indicates a digit (0-9). The string starts with a minus sign if the number is negative.

The precision specifier indicates the desired number of decimal places. If the precision specifier is omitted, the current [NumberFormatInfo.NumberDecimalDigits](#) property supplies the numeric precision.

The result string is affected by the formatting information of the current [NumberFormatInfo](#) object. The following table lists the properties of the [NumberFormatInfo](#) object that control the formatting of the result string.

| NumberFormatInfo property | Description |
|--|--|
| NegativeSign | Defines the string that indicates that a number is negative. |
| NumberDecimalSeparator | Defines the string that separates integral digits from decimal digits. |
| NumberDecimalDigits | Defines the default number of decimal digits. This value can be overridden by using the precision specifier. |

The following example formats a [Double](#) and an [Int32](#) value with the fixed-point format specifier:

C#

```
int integerNumber;
integerNumber = 17843;
Console.WriteLine(integerNumber.ToString("F",
    CultureInfo.InvariantCulture));
// Displays 17843.00

integerNumber = -29541;
Console.WriteLine(integerNumber.ToString("F3",
    CultureInfo.InvariantCulture));
// Displays -29541.000

double doubleNumber;
doubleNumber = 18934.1879;
Console.WriteLine(doubleNumber.ToString("F", CultureInfo.InvariantCulture));
// Displays 18934.19

Console.WriteLine(doubleNumber.ToString("F0",
CultureInfo.InvariantCulture));
// Displays 18934

doubleNumber = -1898300.1987;
Console.WriteLine(doubleNumber.ToString("F1",
```

```

CultureInfo.InvariantCulture));
// Displays -1898300.2

Console.WriteLine(doubleNumber.ToString("F3",
    CultureInfo.CreateSpecificCulture("es-ES")));
// Displays -1898300,199

```

General format specifier (G)

The general ("G") format specifier converts a number to the more compact of either fixed-point or scientific notation, depending on the type of the number and whether a precision specifier is present. The precision specifier defines the maximum number of significant digits that can appear in the result string. If the precision specifier is omitted or zero, the type of the number determines the default precision, as indicated in the following table.

| Numeric type | Default precision |
|-----------------|---|
| Byte or SByte | 3 digits |
| Int16 or UInt16 | 5 digits |
| Int32 or UInt32 | 10 digits |
| Int64 | 19 digits |
| UInt64 | 20 digits |
| BigInteger | Unlimited (same as "R") |
| Half | Smallest round-trippable number of digits to represent the number |
| Single | Smallest round-trippable number of digits to represent the number (in .NET Framework, G7 is the default) |
| Double | Smallest round-trippable number of digits to represent the number (in .NET Framework, G15 is the default) |
| Decimal | Smallest round-trippable number of digits to represent the number |

Fixed-point notation is used if the exponent that would result from expressing the number in scientific notation is greater than -5 and less than the precision specifier; otherwise, scientific notation is used. The result contains a decimal point if required, and trailing zeros after the decimal point are omitted. If the precision specifier is present and

the number of significant digits in the result exceeds the specified precision, the excess trailing digits are removed by rounding.

However, if the number is a [Decimal](#) and the precision specifier is omitted, fixed-point notation is always used and trailing zeros are preserved.

If scientific notation is used, the exponent in the result is prefixed with "E" if the format specifier is "G", or "e" if the format specifier is "g". The exponent contains a minimum of two digits. This differs from the format for scientific notation that is produced by the exponential format specifier, which includes a minimum of three digits in the exponent.

When used with a [Double](#) value, the "G17" format specifier ensures that the original [Double](#) value successfully round-trips. This is because [Double](#) is an IEEE 754-2008-compliant double-precision (`binary64`) floating-point number that gives up to 17 significant digits of precision. On .NET Framework, we recommend its use instead of the "[R](#)" [format specifier](#), since in some cases "R" fails to successfully round-trip double-precision floating point values.

When used with a [Single](#) value, the "G9" format specifier ensures that the original [Single](#) value successfully round-trips. This is because [Single](#) is an IEEE 754-2008-compliant single-precision (`binary32`) floating-point number that gives up to nine significant digits of precision. For performance reasons, we recommend its use instead of the "[R](#)" [format specifier](#).

The result string is affected by the formatting information of the current [NumberFormatInfo](#) object. The following table lists the [NumberFormatInfo](#) properties that control the formatting of the result string.

| NumberFormatInfo property | Description |
|--|--|
| NegativeSign | Defines the string that indicates that a number is negative. |
| NumberDecimalSeparator | Defines the string that separates integral digits from decimal digits. |
| PositiveSign | Defines the string that indicates that an exponent is positive. |

The following example formats assorted floating-point values with the general format specifier:

C#

```
double number;  
  
number = 12345.6789;
```

```

Console.WriteLine(number.ToString("G", CultureInfo.InvariantCulture));
// Displays 12345.6789
Console.WriteLine(number.ToString("G",
                               CultureInfo.CreateSpecificCulture("fr-FR")));
// Displays 12345,6789

Console.WriteLine(number.ToString("G7", CultureInfo.InvariantCulture));
// Displays 12345.68

number = .0000023;
Console.WriteLine(number.ToString("G", CultureInfo.InvariantCulture));
// Displays 2.3E-06
Console.WriteLine(number.ToString("G",
                               CultureInfo.CreateSpecificCulture("fr-FR")));
// Displays 2,3E-06

number = .0023;
Console.WriteLine(number.ToString("G", CultureInfo.InvariantCulture));
// Displays 0.0023

number = 1234;
Console.WriteLine(number.ToString("G2", CultureInfo.InvariantCulture));
// Displays 1.2E+03

number = Math.PI;
Console.WriteLine(number.ToString("G5", CultureInfo.InvariantCulture));
// Displays 3.1416

```

Numeric format specifier (N)

The numeric ("N") format specifier converts a number to a string of the form "-d,ddd,ddd.ddd...", where "-" indicates a negative number symbol if required, "d" indicates a digit (0-9), "," indicates a group separator, and "." indicates a decimal point symbol. The precision specifier indicates the desired number of digits after the decimal point. If the precision specifier is omitted, the number of decimal places is defined by the current [NumberFormatInfo.NumberDecimalDigits](#) property.

The result string is affected by the formatting information of the current [NumberFormatInfo](#) object. The following table lists the [NumberFormatInfo](#) properties that control the formatting of the result string.

| NumberFormatInfo property | Description |
|---------------------------------------|--|
| NegativeSign | Defines the string that indicates that a number is negative. |
| NumberNegativePattern | Defines the format of negative values, and specifies whether the negative sign is represented by parentheses or the NegativeSign |

| NumberFormatInfo property | Description |
|--|--|
| | property. |
| NumberGroupSizes | Defines the number of integral digits that appear between group separators. |
| NumberGroupSeparator | Defines the string that separates groups of integral numbers. |
| NumberDecimalSeparator | Defines the string that separates integral and decimal digits. |
| NumberDecimalDigits | Defines the default number of decimal digits. This value can be overridden by using a precision specifier. |

The following example formats assorted floating-point values with the number format specifier:

C#

```
double dblValue = -12445.6789;
Console.WriteLine(dblValue.ToString("N", CultureInfo.InvariantCulture));
// Displays -12,445.68
Console.WriteLine(dblValue.ToString("N1",
    CultureInfo.CreateSpecificCulture("sv-SE")));
// Displays -12 445,7

int intValue = 123456789;
Console.WriteLine(intValue.ToString("N1", CultureInfo.InvariantCulture));
// Displays 123,456,789.0
```

Percent format specifier (P)

The percent ("P") format specifier multiplies a number by 100 and converts it to a string that represents a percentage. The precision specifier indicates the desired number of decimal places. If the precision specifier is omitted, the default numeric precision supplied by the current [PercentDecimalDigits](#) property is used.

The following table lists the [NumberFormatInfo](#) properties that control the formatting of the returned string.

| NumberFormatInfo property | Description |
|--|--|
| PercentPositivePattern | Defines the placement of the percent symbol for positive values. |
| PercentNegativePattern | Defines the placement of the percent symbol and the negative |

| NumberFormatInfo property | Description |
|--|--|
| | symbol for negative values. |
| NegativeSign | Defines the string that indicates that a number is negative. |
| PercentSymbol | Defines the percent symbol. |
| PercentDecimalDigits | Defines the default number of decimal digits in a percentage value. This value can be overridden by using the precision specifier. |
| PercentDecimalSeparator | Defines the string that separates integral and decimal digits. |
| PercentGroupSeparator | Defines the string that separates groups of integral numbers. |
| PercentGroupSizes | Defines the number of integer digits that appear in a group. |

The following example formats floating-point values with the percent format specifier:

C#

```
double number = .2468013;
Console.WriteLine(number.ToString("P", CultureInfo.InvariantCulture));
// Displays 24.68 %
Console.WriteLine(number.ToString("P",
    CultureInfo.CreateSpecificCulture("hr-HR")));
// Displays 24,68%
Console.WriteLine(number.ToString("P1", CultureInfo.InvariantCulture));
// Displays 24.7 %
```

Round-trip format specifier (R)

The round-trip ("R") format specifier attempts to ensure that a numeric value that is converted to a string is parsed back into the same numeric value. This format is supported only for the [Half](#), [Single](#), [Double](#), and [BigInteger](#) types.

In .NET Framework and in .NET Core versions earlier than 3.0, the "R" format specifier fails to successfully round-trip [Double](#) values in some cases. For both [Double](#) and [Single](#) values, the "R" format specifier offers relatively poor performance. Instead, we recommend that you use the "[G17](#)" format specifier for [Double](#) values and the "[G9](#)" format specifier to successfully round-trip [Single](#) values.

When a [BigInteger](#) value is formatted using this specifier, its string representation contains all the significant digits in the [BigInteger](#) value.

Although you can include a precision specifier, it is ignored. Round trips are given precedence over precision when using this specifier. The result string is affected by the formatting information of the current [NumberFormatInfo](#) object. The following table lists the [NumberFormatInfo](#) properties that control the formatting of the result string.

| NumberFormatInfo property | Description |
|--------------------------------------|--|
| NegativeSign | Defines the string that indicates that a number is negative. |
| NumberDecimalSeparator | Defines the string that separates integral digits from decimal digits. |
| PositiveSign | Defines the string that indicates that an exponent is positive. |

The following example formats a [BigInteger](#) value with the round-trip format specifier.

C#

```
using System;
using System.Numerics;

public class Example
{
    public static void Main()
    {
        var value = BigInteger.Pow(Int64.MaxValue, 2);
        Console.WriteLine(value.ToString("R"));
    }
}
// The example displays the following output:
//      85070591730234615847396907784232501249
```

ⓘ Important

In some cases, [Double](#) values formatted with the "R" standard numeric format string do not successfully round-trip if compiled using the `/platform:x64` or `/platform:anycpu` switches and run on 64-bit systems. See the following paragraph for more information.

To work around the problem of [Double](#) values formatted with the "R" standard numeric format string not successfully round-tripping if compiled using the `/platform:x64` or `/platform:anycpu` switches and run on 64-bit systems, you can format [Double](#) values by using the "G17" standard numeric format string. The following example uses the "R"

format string with a `Double` value that does not round-trip successfully, and also uses the "G17" format string to successfully round-trip the original value:

```
C#  
  
Console.WriteLine("Attempting to round-trip a Double with 'R':");  
double initialValue = 0.6822871999174;  
string valueString = initialValue.ToString("R",  
                                         CultureInfo.InvariantCulture);  
double roundTripped = double.Parse(valueString,  
                                      CultureInfo.InvariantCulture);  
Console.WriteLine("{0:R} = {1:R}: {2}\n",  
                  initialValue, roundTripped,  
                  initialValue.Equals(roundTripped));  
  
Console.WriteLine("Attempting to round-trip a Double with 'G17':");  
string valueString17 = initialValue.ToString("G17",  
                                         CultureInfo.InvariantCulture);  
double roundTripped17 = double.Parse(valueString17,  
                                       CultureInfo.InvariantCulture);  
Console.WriteLine("{0:R} = {1:R}: {2}\n",  
                  initialValue, roundTripped17,  
                  initialValue.Equals(roundTripped17));  
// If compiled to an application that targets anycpu or x64 and run on an  
// x64 system,  
// the example displays the following output:  
//     Attempting to round-trip a Double with 'R':  
//     0.6822871999174 = 0.68228719991740006: False  
//  
//     Attempting to round-trip a Double with 'G17':  
//     0.6822871999174 = 0.6822871999174: True
```

Hexadecimal format specifier (X)

The hexadecimal ("X") format specifier converts a number to a string of hexadecimal digits. The case of the format specifier indicates whether to use uppercase or lowercase characters for hexadecimal digits that are greater than 9. For example, use "X" to produce "ABCDEF", and "x" to produce "abcdef". This format is supported only for integral types.

The precision specifier indicates the minimum number of digits desired in the resulting string. If required, the number is padded with zeros to its left to produce the number of digits given by the precision specifier.

The result string is not affected by the formatting information of the current `NumberFormatInfo` object.

The following example formats `Int32` values with the hexadecimal format specifier.

C#

```
int value;

value = 0x2045e;
Console.WriteLine(value.ToString("x"));
// Displays 2045e
Console.WriteLine(value.ToString("X"));
// Displays 2045E
Console.WriteLine(value.ToString("X8"));
// Displays 0002045E

value = 123456789;
Console.WriteLine(value.ToString("X"));
// Displays 75BCD15
Console.WriteLine(value.ToString("X2"));
// Displays 75BCD15
```

Notes

This section contains additional information about using standard numeric format strings.

Control Panel settings

The settings in the **Regional and Language Options** item in Control Panel influence the result string produced by a formatting operation. Those settings are used to initialize the **NumberFormatInfo** object associated with the current culture, which provides values used to govern formatting. Computers that use different settings generate different result strings.

In addition, if the **CultureInfo(String)** constructor is used to instantiate a new **CultureInfo** object that represents the same culture as the current system culture, any customizations established by the **Regional and Language Options** item in Control Panel will be applied to the new **CultureInfo** object. You can use the **CultureInfo(String, Boolean)** constructor to create a **CultureInfo** object that does not reflect a system's customizations.

NumberFormatInfo properties

Formatting is influenced by the properties of the current **NumberFormatInfo** object, which is provided implicitly by the current culture or explicitly by the **IFormatProvider** parameter of the method that invokes formatting. Specify a **NumberFormatInfo** or **CultureInfo** object for that parameter.

Note

For information about customizing the patterns or strings used in formatting numeric values, see the [NumberFormatInfo](#) class topic.

Integral and floating-point numeric types

Some descriptions of standard numeric format specifiers refer to integral or floating-point numeric types. The integral numeric types are [Byte](#), [SByte](#), [Int16](#), [Int32](#), [Int64](#), [UInt16](#), [UInt32](#), [UInt64](#), and [BigInteger](#). The floating-point numeric types are [Decimal](#), [Half](#), [Single](#), and [Double](#).

Floating-point infinities and NaN

Regardless of the format string, if the value of a [Half](#), [Single](#), or [Double](#) floating-point type is positive infinity, negative infinity, or not a number (NaN), the formatted string is the value of the respective [PositiveInfinitySymbol](#), [NegativeInfinitySymbol](#), or [NaNSymbol](#) property that is specified by the currently applicable [NumberFormatInfo](#) object.

Code example

The following example formats an integral and a floating-point numeric value using the en-US culture and all the standard numeric format specifiers. This example uses two particular numeric types ([Double](#) and [Int32](#)), but would yield similar results for any of the other numeric base types ([Byte](#), [SByte](#), [Int16](#), [Int32](#), [Int64](#), [UInt16](#), [UInt32](#), [UInt64](#), [BigInteger](#), [Decimal](#), [Half](#), and [Single](#)).

C#

```
// Display string representations of numbers for en-us culture
CultureInfo ci = new CultureInfo("en-us");

// Output floating point values
double floating = 10761.937554;
Console.WriteLine("C: {0}",
    floating.ToString("C", ci));           // Displays "C: $10,761.94"
Console.WriteLine("E: {0}",
    floating.ToString("E03", ci));         // Displays "E: 1.076E+004"
Console.WriteLine("F: {0}",
    floating.ToString("F04", ci));         // Displays "F: 10761.9376"
Console.WriteLine("G: {0}",
    floating.ToString("G", ci));          // Displays "G: 10761.937554"
```

```

Console.WriteLine("N: {0}",
    floating.ToString("N03", ci));           // Displays "N: 10,761.938"
Console.WriteLine("P: {0}",
    (floating/10000).ToString("P02", ci)); // Displays "P: 107.62 %"
Console.WriteLine("R: {0}",
    floating.ToString("R", ci));           // Displays "R: 10761.937554"
Console.WriteLine();

// Output integral values
int integral = 8395;
Console.WriteLine("C: {0}",
    integral.ToString("C", ci));           // Displays "C: $8,395.00"
Console.WriteLine("D: {0}",
    integral.ToString("D6", ci));           // Displays "D: 008395"
Console.WriteLine("E: {0}",
    integral.ToString("E03", ci));           // Displays "E: 8.395E+003"
Console.WriteLine("F: {0}",
    integral.ToString("F01", ci));           // Displays "F: 8395.0"
Console.WriteLine("G: {0}",
    integral.ToString("G", ci));            // Displays "G: 8395"
Console.WriteLine("N: {0}",
    integral.ToString("N01", ci));           // Displays "N: 8,395.0"
Console.WriteLine("P: {0}",
    (integral/10000.0).ToString("P02", ci)); // Displays "P: 83.95 %"
Console.WriteLine("X: 0x{0}",
    integral.ToString("X", ci));           // Displays "X: 0x20CB"
Console.WriteLine();

```

See also

- [NumberFormatInfo](#)
- [Custom Numeric Format Strings](#)
- [Formatting Types](#)
- [How to: Pad a Number with Leading Zeros](#)
- [Composite Formatting](#)
- [Sample: .NET Core WinForms Formatting Utility \(C#\)](#)
- [Sample: .NET Core WinForms Formatting Utility \(Visual Basic\)](#)

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For



.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

more information, see [our contributor guide](#).

Custom numeric format strings

Article • 11/20/2021

You can create a custom numeric format string, which consists of one or more custom numeric specifiers, to define how to format numeric data. A custom numeric format string is any format string that is not a [standard numeric format string](#).

Custom numeric format strings are supported by some overloads of the `ToString` method of all numeric types. For example, you can supply a numeric format string to the `ToString(String)` and `ToString(String, IFormatProvider)` methods of the `Int32` type. Custom numeric format strings are also supported by the .NET [composite formatting feature](#), which is used by some `Write` and `WriteLine` methods of the `Console` and `StreamWriter` classes, the `String.Format` method, and the `StringBuilder.AppendFormat` method. [String interpolation](#) feature also supports custom numeric format strings.

Tip

You can download the [Formatting Utility](#), a .NET Core Windows Forms application that lets you apply format strings to either numeric or date and time values and displays the result string. Source code is available for [C#](#) and [Visual Basic](#).

The following table describes the custom numeric format specifiers and displays sample output produced by each format specifier. See the [Notes](#) section for additional information about using custom numeric format strings, and the [Example](#) section for a comprehensive illustration of their use.

| Format specifier | Name | Description | Examples |
|------------------|-------------------|---|--|
| "0" | Zero placeholder | Replaces the zero with the corresponding digit if one is present; otherwise, zero appears in the result string. More information: The "0" Custom Specifier . | 1234.5678 ("00000") -> 01235 0.45678 ("0.00", en-US) -> 0.46 0.45678 ("0.00", fr-FR) -> 0,46 |
| "#" | Digit placeholder | Replaces the "#" symbol with the corresponding digit if one is present; otherwise, no digit appears in the result string. | 1234.5678 ("#####") -> 1235 |

| Format specifier | Name | Description | Examples |
|-------------------------|------------------------------------|--|---|
| | | <p>Note that no digit appears in the result string if the corresponding digit in the input string is a non-significant 0. For example, 0003 ("#####") -> 3.</p> <p>More information: The "#" Custom Specifier.</p> | 0.45678 ("#.##", en-US) -> .46 0.45678 ("#.##", fr-FR) -> ,46 |
| "." | Decimal point | Determines the location of the decimal separator in the result string. | 0.45678 ("0.00", en-US) -> 0.46 |
| | | <p>More information: The "." Custom Specifier.</p> | 0.45678 ("0.00", fr-FR) -> 0,46 |
| "," | Group separator and number scaling | <p>Serves as both a group separator and a number scaling specifier. As a group separator, it inserts a localized group separator character between each group. As a number scaling specifier, it divides a number by 1000 for each comma specified.</p> <p>More information: The "," Custom Specifier.</p> | Group separator specifier: 2147483647 ("##,#", en-US) -> 2,147,483,647 Scaling specifier: 2147483647 ("#,##,", es-ES) -> 2.147.483.647 2147483647 ("#,##,,", en-US) -> 2,147 2147483647 ("#,##,,", es-ES) -> 2.147 |
| "%" | Percentage placeholder | <p>Multiplies a number by 100 and inserts a localized percentage symbol in the result string.</p> <p>More information: The "%" Custom Specifier.</p> | 0.3697 ("%#0.00", en-US) -> %36.97 0.3697 ("%#0.00", el-GR) -> %36,97 0.3697 ("##.0%", en-US) -> 37.0 % 0.3697 ("##.0") |

| Format specifier | Name | Description | Examples |
|------------------|--------------------------|--|---|
| | | | %", el-GR) -> 37,0 % |
| "%o" | Per mille placeholder | Multiplies a number by 1000 and inserts a localized per mille symbol in the result string. More information: The "%o" Custom Specifier . | 0.03697 ("0.00%", en-US) -> 36.97% |
| | | | 0.03697 ("0.00%", ru-RU) -> 36,97% |
| "E0" | Exponential notation | If followed by at least one 0 (zero), formats the result using exponential notation. The case of "E" or "e" indicates the case of the exponent symbol in the result string. The number of zeros following the "E" or "e" character determines the minimum number of digits in the exponent. A plus sign (+) indicates that a sign character always precedes the exponent. A minus sign (-) indicates that a sign character precedes only negative exponents. | 987654 ("#0.0e0") -> 98.8e4 |
| "E-0" | | | 1503.92311 ("0.0##e+00") -> 1.504e+03 |
| "e0" | | | |
| "e+0" | | | 1.8901385E-16 ("0.0e+00") -> |
| "e-0" | | More information: The "E" and "e" Custom Specifiers . | 1.9e-16 |
| "\" | Escape character | Causes the next character to be interpreted as a literal rather than as a custom format specifier. More information: The "\" Escape Character . | 987654 ("###00\#") -> #987654# |
| 'string' | Literal string delimiter | Indicates that the enclosed characters should be copied to the result string unchanged. | 68 ("# 'degrees'") -> 68 degrees |
| "string" | | More information: Character literals . | 68 ("#' degrees'") -> 68 degrees |
| ; | Section separator | Defines sections with separate format strings for positive, negative, and zero numbers. More information: The ";" Section Separator . | 12.345 ("#0.0#; (#0.0#);-\0-") -> 12.35 0 ("#0.0#; (#0.0#);-\0-") -> -0- |
| | | | -12.345 ("#0.0#; (#0.0#);-\0-") -> (12.35) |

| Format specifier | Name | Description | Examples |
|--|----------------------|---|---|
| | | | 12.345 ("#0.0#; (#0.0#)") -> 12.35 |
| | | | 0 ("#0.0#; (#0.0#)") -> 0.0 |
| | | | -12.345 ("#0.0#; (#0.0#)") -> (12.35) |
| Other | All other characters | The character is copied to the result string unchanged. | 68 ("# °") -> 68° |
| More information: Character literals . | | | |

The following sections provide detailed information about each of the custom numeric format specifiers.

ⓘ Note

Some of the C# examples in this article run in the [Try.NET](#) inline code runner and playground. Select the **Run** button to run an example in an interactive window. Once you execute the code, you can modify it and run the modified code by selecting **Run** again. The modified code either runs in the interactive window or, if compilation fails, the interactive window displays all C# compiler error messages.

The "0" custom specifier

The "0" custom format specifier serves as a zero-placeholder symbol. If the value that is being formatted has a digit in the position where the zero appears in the format string, that digit is copied to the result string; otherwise, a zero appears in the result string. The position of the leftmost zero before the decimal point and the rightmost zero after the decimal point determines the range of digits that are always present in the result string.

The "00" specifier causes the value to be rounded to the nearest digit preceding the decimal, where rounding away from zero is always used. For example, formatting 34.5 with "00" would result in the value 35.

The following example displays several values that are formatted by using custom format strings that include zero placeholders.

C#

```
double value;

value = 123;
Console.WriteLine(value.ToString("00000"));
Console.WriteLine(String.Format("{0:00000}", value));
// Displays 00123

value = 1.2;
Console.WriteLine(value.ToString("0.00", CultureInfo.InvariantCulture));
Console.WriteLine(String.Format(CultureInfo.InvariantCulture,
    "{0:0.00}", value));
// Displays 1.20

Console.WriteLine(value.ToString("00.00", CultureInfo.InvariantCulture));
Console.WriteLine(String.Format(CultureInfo.InvariantCulture,
    "{0:00.00}", value));
// Displays 01.20

CultureInfo daDK = CultureInfo.CreateSpecificCulture("da-DK");
Console.WriteLine(value.ToString("00.00", daDK));
Console.WriteLine(String.Format(daDK, "{0:00.00}", value));
// Displays 01,20

value = .56;
Console.WriteLine(value.ToString("0.0", CultureInfo.InvariantCulture));
Console.WriteLine(String.Format(CultureInfo.InvariantCulture,
    "{0:0.0}", value));
// Displays 0.6

value = 1234567890;
Console.WriteLine(value.ToString("0,0", CultureInfo.InvariantCulture));
Console.WriteLine(String.Format(CultureInfo.InvariantCulture,
    "{0:0,0}", value));
// Displays 1,234,567,890

CultureInfo elGR = CultureInfo.CreateSpecificCulture("el-GR");
Console.WriteLine(value.ToString("0,0", elGR));
Console.WriteLine(String.Format(elGR, "{0:0,0}", value));
// Displays 1.234.567.890

value = 1234567890.123456;
Console.WriteLine(value.ToString("0,0.0", CultureInfo.InvariantCulture));
Console.WriteLine(String.Format(CultureInfo.InvariantCulture,
    "{0:0,0.0}", value));
// Displays 1,234,567,890.1

value = 1234.567890;
Console.WriteLine(value.ToString("0,0.00", CultureInfo.InvariantCulture));
Console.WriteLine(String.Format(CultureInfo.InvariantCulture,
```

```
        "{0:0,0.00}", value));  
// Displays 1,234.57
```

[Back to table](#)

The "#" custom specifier

The "#" custom format specifier serves as a digit-placeholder symbol. If the value that is being formatted has a digit in the position where the "#" symbol appears in the format string, that digit is copied to the result string. Otherwise, nothing is stored in that position in the result string.

Note that this specifier never displays a zero that is not a significant digit, even if zero is the only digit in the string. It will display zero only if it is a significant digit in the number that is being displayed.

The "##" format string causes the value to be rounded to the nearest digit preceding the decimal, where rounding away from zero is always used. For example, formatting 34.5 with "##" would result in the value 35.

The following example displays several values that are formatted by using custom format strings that include digit placeholders.

```
C#  
  
double value;  
  
value = 1.2;  
Console.WriteLine(value.ToString("#.##", CultureInfo.InvariantCulture));  
Console.WriteLine(String.Format(CultureInfo.InvariantCulture,  
        "{0:#.##}", value));  
// Displays 1.2  
  
value = 123;  
Console.WriteLine(value.ToString("#####"));  
Console.WriteLine(String.Format("{0:#####}", value));  
// Displays 123  
  
value = 123456;  
Console.WriteLine(value.ToString("[##-##-##]"));  
Console.WriteLine(String.Format("{0:[##-##-##]}", value));  
// Displays [12-34-56]  
  
value = 1234567890;  
Console.WriteLine(value.ToString("#"));  
Console.WriteLine(String.Format("{0:#}", value));  
// Displays 1234567890
```

```
Console.WriteLine(value.ToString("###) ###-###"));
Console.WriteLine(String.Format("{0:(###) ###-###}", value));
// Displays (123) 456-7890
```

To return a result string in which absent digits or leading zeroes are replaced by spaces, use the [composite formatting feature](#) and specify a field width, as the following example illustrates.

C#

```
using System;

public class SpaceOrDigit
{
    public static void Main()
    {
        Double value = .324;
        Console.WriteLine("The value is: '{0,5:#.###}'", value);
    }
}
// The example displays the following output if the current culture
// is en-US:
//      The value is: '.324'
```

[Back to table](#)

The "." custom specifier

The "." custom format specifier inserts a localized decimal separator into the result string. The first period in the format string determines the location of the decimal separator in the formatted value; any additional periods are ignored. If the format specifier ends with a "." only the significant digits are formatted into the result string.

The character that is used as the decimal separator in the result string is not always a period; it is determined by the [NumberDecimalSeparator](#) property of the [NumberFormatInfo](#) object that controls formatting.

The following example uses the "." format specifier to define the location of the decimal point in several result strings.

C#

```
double value;

value = 1.2;
Console.WriteLine(value.ToString("0.00", CultureInfo.InvariantCulture));
Console.WriteLine(String.Format(CultureInfo.InvariantCulture,
```

```

        "{0:0.00}", value));
// Displays 1.20

Console.WriteLine(value.ToString("00.00", CultureInfo.InvariantCulture));
Console.WriteLine(String.Format(CultureInfo.InvariantCulture,
                               "{0:00.00}", value));
// Displays 01.20

Console.WriteLine(value.ToString("00.00",
                               CultureInfo.CreateSpecificCulture("da-DK")));
Console.WriteLine(String.Format(CultureInfo.CreateSpecificCulture("da-DK"),
                               "{0:00.00}", value));
// Displays 01,20

value = .086;
Console.WriteLine(value.ToString("#0.##%", CultureInfo.InvariantCulture));
Console.WriteLine(String.Format(CultureInfo.InvariantCulture,
                               "{0:#0.##%}", value));
// Displays 8.6%

value = 86000;
Console.WriteLine(value.ToString("0.###E+0", CultureInfo.InvariantCulture));
Console.WriteLine(String.Format(CultureInfo.InvariantCulture,
                               "{0:0.###E+0}", value));
// Displays 8.6E+4

```

[Back to table](#)

The "," custom specifier

The "," character serves as both a group separator and a number scaling specifier.

- Group separator: If one or more commas are specified between two digit placeholders (0 or #) that format the integral digits of a number, a group separator character is inserted between each number group in the integral part of the output.

The [NumberGroupSeparator](#) and [NumberGroupSizes](#) properties of the current [NumberFormatInfo](#) object determine the character used as the number group separator and the size of each number group. For example, if the string "#,#" and the invariant culture are used to format the number 1000, the output is "1,000".

- Number scaling specifier: If one or more commas are specified immediately to the left of the explicit or implicit decimal point, the number to be formatted is divided by 1000 for each comma. For example, if the string "0,," is used to format the number 100 million, the output is "100".

You can use group separator and number scaling specifiers in the same format string. For example, if the string "#,0," and the invariant culture are used to format the number one billion, the output is "1,000".

The following example illustrates the use of the comma as a group separator.

```
C#  
  
double value = 1234567890;  
Console.WriteLine(value.ToString("#,#", CultureInfo.InvariantCulture));  
Console.WriteLine(String.Format(CultureInfo.InvariantCulture,  
        "{0:#,#}", value));  
// Displays 1,234,567,890  
  
Console.WriteLine(value.ToString("#,##0,,", CultureInfo.InvariantCulture));  
Console.WriteLine(String.Format(CultureInfo.InvariantCulture,  
        "{0:#,##0,,}", value));  
// Displays 1,235
```

The following example illustrates the use of the comma as a specifier for number scaling.

```
C#  
  
double value = 1234567890;  
Console.WriteLine(value.ToString("#,,", CultureInfo.InvariantCulture));  
Console.WriteLine(String.Format(CultureInfo.InvariantCulture,  
        "{0:#,,}", value));  
// Displays 1235  
  
Console.WriteLine(value.ToString("#,,,", CultureInfo.InvariantCulture));  
Console.WriteLine(String.Format(CultureInfo.InvariantCulture,  
        "{0:#,,,}", value));  
// Displays 1  
  
Console.WriteLine(value.ToString("#,##0,,", CultureInfo.InvariantCulture));  
Console.WriteLine(String.Format(CultureInfo.InvariantCulture,  
        "{0:#,##0,,}", value));  
// Displays 1,235
```

[Back to table](#)

The "%" custom specifier

A percent sign (%) in a format string causes a number to be multiplied by 100 before it is formatted. The localized percent symbol is inserted in the number at the location

where the % appears in the format string. The percent character used is defined by the [PercentSymbol](#) property of the current [NumberFormatInfo](#) object.

The following example defines several custom format strings that include the "%" custom specifier.

C#

```
double value = .086;
Console.WriteLine(value.ToString("#0.##%", CultureInfo.InvariantCulture));
Console.WriteLine(String.Format(CultureInfo.InvariantCulture,
    "{0:#0.##%}", value));
// Displays 8.6%
```

[Back to table](#)

The "%o" custom specifier

A per mille character (%o or \u2030) in a format string causes a number to be multiplied by 1000 before it is formatted. The appropriate per mille symbol is inserted in the returned string at the location where the %o symbol appears in the format string. The per mille character used is defined by the [NumberFormatInfo.PerMilleSymbol](#) property of the object that provides culture-specific formatting information.

The following example defines a custom format string that includes the "%o" custom specifier.

C#

```
double value = .00354;
string perMilleFmt = "#0.## " + '\u2030';
Console.WriteLine(value.ToString(perMilleFmt,
CultureInfo.InvariantCulture));
Console.WriteLine(String.Format(CultureInfo.InvariantCulture,
    "{0:" + perMilleFmt + "}", value));
// Displays 3.54%
```

[Back to table](#)

The "E" and "e" custom specifiers

If any of the strings "E", "E+", "E-", "e", "e+", or "e-" are present in the format string and are followed immediately by at least one zero, the number is formatted by using scientific notation with an "E" or "e" inserted between the number and the exponent.

The number of zeros following the scientific notation indicator determines the minimum number of digits to output for the exponent. The "E+" and "e+" formats indicate that a plus sign or minus sign should always precede the exponent. The "E", "E-", "e", or "e-" formats indicate that a sign character should precede only negative exponents.

The following example formats several numeric values using the specifiers for scientific notation.

```
C#
```

```
double value = 86000;
Console.WriteLine(value.ToString("0.###E+0", CultureInfo.InvariantCulture));
Console.WriteLine(String.Format(CultureInfo.InvariantCulture,
    "{0:0.###E+0}", value));
// Displays 8.6E+4

Console.WriteLine(value.ToString("0.###E+000",
CultureInfo.InvariantCulture));
Console.WriteLine(String.Format(CultureInfo.InvariantCulture,
    "{0:0.###E+000}", value));
// Displays 8.6E+004

Console.WriteLine(value.ToString("0.###E-000",
CultureInfo.InvariantCulture));
Console.WriteLine(String.Format(CultureInfo.InvariantCulture,
    "{0:0.###E-000}", value));
// Displays 8.6E004
```

[Back to table](#)

The "\ escape character

The "#", "0", ".", ",", "%", and "%o" symbols in a format string are interpreted as format specifiers rather than as literal characters. Depending on their position in a custom format string, the uppercase and lowercase "E" as well as the + and - symbols may also be interpreted as format specifiers.

To prevent a character from being interpreted as a format specifier, you can precede it with a backslash, which is the escape character. The escape character signifies that the following character is a character literal that should be included in the result string unchanged.

To include a backslash in a result string, you must escape it with another backslash (\\\).

 **Note**

Some compilers, such as the C++ and C# compilers, may also interpret a single backslash character as an escape character. To ensure that a string is interpreted correctly when formatting, you can use the verbatim string literal character (the @ character) before the string in C#, or add another backslash character before each backslash in C# and C++. The following C# example illustrates both approaches.

The following example uses the escape character to prevent the formatting operation from interpreting the "#", "0", and "\" characters as either escape characters or format specifiers. The C# examples uses an additional backslash to ensure that a backslash is interpreted as a literal character.

C#

```
int value = 123;
Console.WriteLine(value.ToString("\#\#\#\#\##0 dollars and \0\0 cents
\#\#\#\#"));
Console.WriteLine(String.Format("{0:\#\#\#\#\##0 dollars and \0\0 cents
\#\#\#\#}", value));
// Displays ### 123 dollars and 00 cents ###

Console.WriteLine(value.ToString(@"#\#\#\##0 dollars and \0\0 cents
#\#\#\#"));
Console.WriteLine(String.Format(@"{0:#\#\#\##0 dollars and \0\0 cents
#\#\#\#}", value));
// Displays ### 123 dollars and 00 cents ###

Console.WriteLine(value.ToString("\\\\\\\\\\##0 dollars and \0\0 cents
\\\\\\\\\\\\\\\\"));
Console.WriteLine(String.Format("{0:\\\\\\\\\\##0 dollars and \0\0 cents
\\\\\\\\\\\\\\\\}", value));
// Displays \\ 123 dollars and 00 cents \\

Console.WriteLine(value.ToString(@"\\\\\\##0 dollars and \0\0 cents
\\\\\\"));
Console.WriteLine(String.Format(@"{0:\\\\\\##0 dollars and \0\0 cents
\\\\\\}", value));
// Displays \\ 123 dollars and 00 cents \\
```

[Back to table](#)

The ";" section separator

The semicolon (;) is a conditional format specifier that applies different formatting to a number depending on whether its value is positive, negative, or zero. To produce this behavior, a custom format string can contain up to three sections separated by semicolons. These sections are described in the following table.

| Number of sections | Description |
|--------------------|---|
| One section | The format string applies to all values. |
| Two sections | <p>The first section applies to positive values and zeros, and the second section applies to negative values.</p> <p>If the number to be formatted is negative, but becomes zero after rounding according to the format in the second section, the resulting zero is formatted according to the first section.</p> |
| Three sections | <p>The first section applies to positive values, the second section applies to negative values, and the third section applies to zeros.</p> <p>The second section can be left empty (by having nothing between the semicolons), in which case the first section applies to all nonzero values.</p> <p>If the number to be formatted is nonzero, but becomes zero after rounding according to the format in the first or second section, the resulting zero is formatted according to the third section.</p> |

Section separators ignore any preexisting formatting associated with a number when the final value is formatted. For example, negative values are always displayed without a minus sign when section separators are used. If you want the final formatted value to have a minus sign, you should explicitly include the minus sign as part of the custom format specifier.

The following example uses the ";" format specifier to format positive, negative, and zero numbers differently.

C#

```
double posValue = 1234;
double negValue = -1234;
double zeroValue = 0;

string fmt2 = "##;(#)";
string fmt3 = "##;(#);**Zero**";

Console.WriteLine(posValue.ToString(fmt2));
Console.WriteLine(String.Format("{0:" + fmt2 + "}", posValue));
// Displays 1234
```

```
Console.WriteLine(negValue.ToString(fmt2));
Console.WriteLine(String.Format("{0:" + fmt2 + "}", negValue));
// Displays (1234)

Console.WriteLine(zeroValue.ToString(fmt3));
Console.WriteLine(String.Format("{0:" + fmt3 + "}", zeroValue));
// Displays **Zero**
```

[Back to table](#)

Character literals

Format specifiers that appear in a custom numeric format string are always interpreted as formatting characters and never as literal characters. This includes the following characters:

- 0
- #
- %
- %o
- '
- \
- .
- ,
- E or e, depending on its position in the format string.

All other characters are always interpreted as character literals and, in a formatting operation, are included in the result string unchanged. In a parsing operation, they must match the characters in the input string exactly; the comparison is case-sensitive.

The following example illustrates one common use of literal character units (in this case, thousands):

```
C#
double n = 123.8;
Console.WriteLine($"{n:#,##0.0K}");
// The example displays the following output:
//      123.8K
```

There are two ways to indicate that characters are to be interpreted as literal characters and not as formatting characters, so that they can be included in a result string or successfully parsed in an input string:

- By escaping a formatting character. For more information, see [The "\" escape character](#).
- By enclosing the entire literal string in quotation apostrophes.

The following example uses both approaches to include reserved characters in a custom numeric format string.

```
C#  
  
double n = 9.3;  
Console.WriteLine(${@n:##.0%});  
Console.WriteLine(${@n:\##'}");  
Console.WriteLine(${@n:\\##\\});  
Console.WriteLine();  
Console.WriteLine("${n:##.0%'");  
Console.WriteLine(${@n:'\##'\});  
// The example displays the following output:  
//      9.3%  
//      '9'  
//      \9\  
//  
//      9.3%  
//      \9\
```

Notes

Floating-Point infinities and NaN

Regardless of the format string, if the value of a [Half](#), [Single](#), or [Double](#) floating-point type is positive infinity, negative infinity, or not a number (NaN), the formatted string is the value of the respective [PositiveInfinitySymbol](#), [NegativeInfinitySymbol](#), or [NaNSymbol](#) property specified by the currently applicable [NumberFormatInfo](#) object.

Control Panel settings

The settings in the [Regional and Language Options](#) item in Control Panel influence the result string produced by a formatting operation. Those settings are used to initialize the [NumberFormatInfo](#) object associated with the current culture, and the current culture provides values used to govern formatting. Computers that use different settings generate different result strings.

In addition, if you use the [CultureInfo\(String\)](#) constructor to instantiate a new [CultureInfo](#) object that represents the same culture as the current system culture, any

customizations established by the [Regional and Language Options](#) item in Control Panel will be applied to the new [CultureInfo](#) object. You can use the [CultureInfo\(String, Boolean\)](#) constructor to create a [CultureInfo](#) object that does not reflect a system's customizations.

Rounding and fixed-point format strings

For fixed-point format strings (that is, format strings that do not contain scientific notation format characters), numbers are rounded to as many decimal places as there are digit placeholders to the right of the decimal point. If the format string does not contain a decimal point, the number is rounded to the nearest integer. If the number has more digits than there are digit placeholders to the left of the decimal point, the extra digits are copied to the result string immediately before the first digit placeholder.

[Back to table](#)

Example

The following example demonstrates two custom numeric format strings. In both cases, the digit placeholder (#) displays the numeric data, and all other characters are copied to the result string.

C#

```
double number1 = 1234567890;
string value1 = number1.ToString("(###) ###-####");
Console.WriteLine(value1);

int number2 = 42;
string value2 = number2.ToString("My Number = #");
Console.WriteLine(value2);
// The example displays the following output:
//      (123) 456-7890
//      My Number = 42
```

[Back to table](#)

See also

- [System.Globalization.NumberFormatInfo](#)
- [Formatting Types](#)
- [Standard Numeric Format Strings](#)
- [How to: Pad a Number with Leading Zeros](#)

- Sample: .NET Core WinForms Formatting Utility (C#)
- Sample: .NET Core WinForms Formatting Utility (Visual Basic)

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

Standard date and time format strings

Article • 05/02/2023

A standard date and time format string uses a single character as the format specifier to define the text representation of a [DateTime](#) or a [DateTimeOffset](#) value. Any date and time format string that contains more than one character, including white space, is interpreted as a [custom date and time format string](#). A standard or custom format string can be used in two ways:

- To define the string that results from a formatting operation.
- To define the text representation of a date and time value that can be converted to a [DateTime](#) or [DateTimeOffset](#) value by a parsing operation.

Tip

You can download the [Formatting Utility](#), a .NET Windows Forms application that lets you apply format strings to either numeric or date and time values and display the result string. Source code is available for [C#](#) and [Visual Basic](#).

Note

Some of the C# examples in this article run in the [Try.NET](#) inline code runner and playground. Select the **Run** button to run an example in an interactive window. Once you execute the code, you can modify it and run the modified code by selecting **Run** again. The modified code either runs in the interactive window or, if compilation fails, the interactive window displays all C# compiler error messages.

The **local time zone** of the [Try.NET](#) inline code runner and playground is Coordinated Universal Time, or UTC. This may affect the behavior and the output of examples that illustrate the [DateTime](#), [DateTimeOffset](#), and [TimeZoneInfo](#) types and their members.

Table of format specifiers

The following table describes the standard date and time format specifiers. Unless otherwise noted, a particular standard date and time format specifier produces an identical string representation regardless of whether it is used with a [DateTime](#) or a

`DateTimeOffset` value. See [Control Panel Settings](#) and [DateTimeFormatInfo Properties](#) for additional information about using standard date and time format strings.

| Format specifier | Description | Examples |
|------------------|---|---|
| "d" | Short date pattern. More information: The short date ("d") format specifier . | 2009-06-15T13:45:30 -> 6/15/2009 (en-US) 2009-06-15T13:45:30 -> 15/06/2009 (fr-FR) 2009-06-15T13:45:30 -> 2009/06/15 (ja-JP) |
| "D" | Long date pattern. More information: The long date ("D") format specifier . | 2009-06-15T13:45:30 -> Monday, June 15, 2009 (en-US) 2009-06-15T13:45:30 -> понедельник, 15 июня 2009 г. (ru-RU) 2009-06-15T13:45:30 -> Montag, 15. Juni 2009 (de-DE) |
| "f" | Full date/time pattern (short time). More information: The full date short time ("f") format specifier . | 2009-06-15T13:45:30 -> Monday, June 15, 2009 1:45 PM (en-US) 2009-06-15T13:45:30 -> den 15 juni 2009 13:45 (sv-SE) 2009-06-15T13:45:30 -> Δευτέρα, 15 Ιουνίου 2009 1:45 μμ (el-GR) |
| "F" | Full date/time pattern (long time). More information: The full date long time ("F") format specifier . | 2009-06-15T13:45:30 -> Monday, June 15, 2009 1:45:30 PM (en-US) 2009-06-15T13:45:30 -> den 15 juni 2009 13:45:30 (sv-SE) 2009-06-15T13:45:30 -> Δευτέρα, 15 Ιουνίου 2009 1:45:30 μμ (el-GR) |
| "g" | General date/time pattern (short time). More information: The general date short time ("g") format specifier . | 2009-06-15T13:45:30 -> 6/15/2009 1:45 PM (en-US) 2009-06-15T13:45:30 -> 15/06/2009 13:45 (es-ES) 2009-06-15T13:45:30 -> 2009/6/15 13:45 (zh-CN) |
| "G" | General date/time pattern (long time). | 2009-06-15T13:45:30 -> 6/15/2009 1:45:30 PM (en-US) |

| Format specifier | Description | Examples |
|-------------------------|--|---|
| | More information: The general date long time ("G") format specifier. | 2009-06-15T13:45:30 -> 15/06/2009 13:45:30 (es-ES) 2009-06-15T13:45:30 -> 2009/6/15 13:45:30 (zh-CN) |
| "M", "m" | Month/day pattern. More information: The month ("M", "m") format specifier. | 2009-06-15T13:45:30 -> June 15 (en-US) 2009-06-15T13:45:30 -> 15. juni (da-DK) 2009-06-15T13:45:30 -> 15 Juni (id-ID) |
| "O", "o" | round-trip date/time pattern. More information: The round-trip ("O", "o") format specifier. | DateTime values: 2009-06-15T13:45:30 (DateTimeKind.Local) --> 2009-06-15T13:45:30.0000000-07:00 2009-06-15T13:45:30 (DateTimeKind.Utc) --> 2009-06-15T13:45:30.0000000Z 2009-06-15T13:45:30 (DateTimeKind.Unspecified) --> 2009-06-15T13:45:30.0000000 |
| | | DateTimeOffset values: 2009-06-15T13:45:30-07:00 --> 2009-06-15T13:45:30.0000000-07:00 |
| "R", "r" | RFC1123 pattern. More information: The RFC1123 ("R", "r") format specifier. | DateTimeOffset input: 2009-06-15T13:45:30 -> Mon, 15 Jun 2009 20:45:30 GMT DateTime input: 2009-06-15T13:45:30 -> Mon, 15 Jun 2009 13:45:30 GMT |
| "s" | Sortable date/time pattern. More information: The sortable ("s") format specifier. | 2009-06-15T13:45:30 (DateTimeKind.Local) -> 2009-06-15T13:45:30 2009-06-15T13:45:30 (DateTimeKind.Utc) -> 2009-06-15T13:45:30 |
| "t" | Short time pattern. More information: The short time ("t") format specifier. | 2009-06-15T13:45:30 -> 1:45 PM (en-US) 2009-06-15T13:45:30 -> 13:45 (hr-HR) 2009-06-15T13:45:30 -> 01:45 ρ (ar-EG) |

| Format specifier | Description | Examples |
|----------------------------|--|---|
| "T" | <p>Long time pattern.</p> <p>More information: The long time ("T") format specifier.</p> | <p>2009-06-15T13:45:30 -> 1:45:30 PM (en-US)</p> <p>2009-06-15T13:45:30 -> 13:45:30 (hr-HR)</p> <p>2009-06-15T13:45:30 -> 01:45:30 ρ (ar-EG)</p> |
| "u" | <p>Universal sortable date/time pattern.</p> <p>More information: The universal sortable ("u") format specifier.</p> | <p>With a DateTime value: 2009-06-15T13:45:30 -> 2009-06-15 13:45:30Z</p> <p>With a DateTimeOffset value: 2009-06-15T13:45:30 -> 2009-06-15 20:45:30Z</p> |
| "U" | <p>Universal full date/time pattern.</p> <p>More information: The universal full ("U") format specifier.</p> | <p>2009-06-15T13:45:30 -> Monday, June 15, 2009 8:45:30 PM (en-US)</p> <p>2009-06-15T13:45:30 -> den 15 juni 2009 20:45:30 (sv-SE)</p> <p>2009-06-15T13:45:30 -> Δευτέρα, 15 Ιουνίου 2009 8:45:30 μμ (el-GR)</p> |
| "Y", "y" | <p>Year month pattern.</p> <p>More information: The year month ("Y") format specifier.</p> | <p>2009-06-15T13:45:30 -> June 2009 (en-US)</p> <p>2009-06-15T13:45:30 -> juni 2009 (da-DK)</p> <p>2009-06-15T13:45:30 -> Juni 2009 (id-ID)</p> |
| Any other single character | Unknown specifier. | Throws a run-time FormatException . |

How standard format strings work

In a formatting operation, a standard format string is simply an alias for a custom format string. The advantage of using an alias to refer to a custom format string is that, although the alias remains invariant, the custom format string itself can vary. This is important because the string representations of date and time values typically vary by culture. For example, the "d" standard format string indicates that a date and time value is to be displayed using a short date pattern. For the invariant culture, this pattern is "MM/dd/yyyy". For the fr-FR culture, it is "dd/MM/yyyy". For the ja-JP culture, it is "yyyy/MM/dd".

If a standard format string in a formatting operation maps to a particular culture's custom format string, your application can define the specific culture whose custom

format strings are used in one of these ways:

- You can use the default (or current) culture. The following example displays a date using the current culture's short date format. In this case, the current culture is en-US.

C#

```
// Display using current (en-us) culture's short date format
DateTime thisDate = new DateTime(2008, 3, 15);
Console.WriteLine(thisDate.ToString("d"));           // Displays
3/15/2008
```

- You can pass a [CultureInfo](#) object representing the culture whose formatting is to be used to a method that has an [IFormatProvider](#) parameter. The following example displays a date using the short date format of the pt-BR culture.

C#

```
// Display using pt-BR culture's short date format
DateTime thisDate = new DateTime(2008, 3, 15);
CultureInfo culture = new CultureInfo("pt-BR");
Console.WriteLine(thisDate.ToString("d", culture)); // Displays
15/3/2008
```

- You can pass a [DateTimeFormatInfo](#) object that provides formatting information to a method that has an [IFormatProvider](#) parameter. The following example displays a date using the short date format from a [DateTimeFormatInfo](#) object for the hr-HR culture.

C#

```
// Display using date format information from hr-HR culture
DateTime thisDate = new DateTime(2008, 3, 15);
DateTimeFormatInfo fmt = (new CultureInfo("hr-HR")).DateTimeFormat;
Console.WriteLine(thisDate.ToString("d", fmt));      // Displays
15.3.2008
```

ⓘ Note

For information about customizing the patterns or strings used in formatting date and time values, see the [NumberFormatInfo](#) class topic.

In some cases, the standard format string serves as a convenient abbreviation for a longer custom format string that is invariant. Four standard format strings fall into this category: "O" (or "o"), "R" (or "r"), "s", and "u". These strings correspond to custom format strings defined by the invariant culture. They produce string representations of date and time values that are intended to be identical across cultures. The following table provides information on these four standard date and time format strings.

| Standard format string | Defined by DateTimeFormatInfo.InvariantInfo property | Custom format string |
|-------------------------------|--|--|
| "O" or "o" | None | yyyy'-'MM'- 'dd'T'HH':'mm':'ss'.ffffffffffK |
| "R" or "r" | RFC1123Pattern | ddd, dd MMM yyyy HH':'mm':'ss 'GMT' |
| "s" | SortableDateTimePattern | yyyy'-'MM'-'dd'T'HH':'mm':'ss |
| "u" | UniversalSortableDateTimePattern | yyyy'-'MM'-'dd HH':'mm':'ss'Z' |

Standard format strings can also be used in parsing operations with the [DateTime.ParseExact](#) or [DateTimeOffset.ParseExact](#) methods, which require an input string to exactly conform to a particular pattern for the parse operation to succeed. Many standard format strings map to multiple custom format strings, so a date and time value can be represented in a variety of formats and the parse operation will still succeed. You can determine the custom format string or strings that correspond to a standard format string by calling the [DateTimeFormatInfo.GetAllDateTimePatterns\(Char\)](#) method. The following example displays the custom format strings that map to the "d" (short date pattern) standard format string.

C#

```
using System;
using System.Globalization;

public class Example
{
    public static void Main()
    {
        Console.WriteLine("'d' standard format string:");
        foreach (var customString in
DateTimeFormatInfo.CurrentInfo.GetAllDateTimePatterns('d'))
            Console.WriteLine(" {0}", customString);
    }
}
// The example displays the following output:
//      'd' standard format string:
```

```
//      M/d/yyyy  
//      M/d/yy  
//      MM/dd/yy  
//      MM/dd/yyyy  
//      yy/MM/dd  
//      yyyy-MM-dd  
//      dd-MMM-yy
```

The following sections describe the standard format specifiers for [DateTime](#) and [DateTimeOffset](#) values.

Date formats

This group includes the following formats:

- The short date ("d") format specifier
- The long date ("D") format specifier

The short date ("d") format specifier

The "d" standard format specifier represents a custom date and time format string that is defined by a specific culture's [DateTimeFormatInfo.ShortDatePattern](#) property. For example, the custom format string that is returned by the [ShortDatePattern](#) property of the invariant culture is "MM/dd/yyyy".

The following table lists the [DateTimeFormatInfo](#) object properties that control the formatting of the returned string.

| Property | Description |
|----------------------------------|--|
| ShortDatePattern | Defines the overall format of the result string. |
| DateSeparator | Defines the string that separates the year, month, and day components of a date. |

The following example uses the "d" format specifier to display a date and time value.

C#

```
DateTime date1 = new DateTime(2008,4, 10);  
Console.WriteLine(date1.ToString("d", DateTimeFormatInfo.InvariantInfo));  
// Displays 04/10/2008  
Console.WriteLine(date1.ToString("d",  
        CultureInfo.CreateSpecificCulture("en-US")));  
// Displays 4/10/2008  
Console.WriteLine(date1.ToString("d",
```

```

        CultureInfo.CreateSpecificCulture("en-NZ")));
// Displays 10/04/2008
Console.WriteLine(date1.ToString("d",
        CultureInfo.CreateSpecificCulture("de-DE")));
// Displays 10.04.2008

```

[Back to table](#)

The long date ("D") format specifier

The "D" standard format specifier represents a custom date and time format string that is defined by the current [DateTimeFormatInfo.LongDatePattern](#) property. For example, the custom format string for the invariant culture is "dddd, dd MMMM yyyy".

The following table lists the properties of the [DateTimeFormatInfo](#) object that control the formatting of the returned string.

| Property | Description |
|---------------------------------|---|
| LongDatePattern | Defines the overall format of the result string. |
| DayNames | Defines the localized day names that can appear in the result string. |
| MonthNames | Defines the localized month names that can appear in the result string. |

The following example uses the "D" format specifier to display a date and time value.

C#

```

DateTime date1 = new DateTime(2008, 4, 10);
Console.WriteLine(date1.ToString("D",
        CultureInfo.CreateSpecificCulture("en-US")));
// Displays Thursday, April 10, 2008
Console.WriteLine(date1.ToString("D",
        CultureInfo.CreateSpecificCulture("pt-BR")));
// Displays quinta-feira, 10 de abril de 2008
Console.WriteLine(date1.ToString("D",
        CultureInfo.CreateSpecificCulture("es-MX")));
// Displays jueves, 10 de abril de 2008

```

[Back to table](#)

Date and time formats

This group includes the following formats:

- The full date short time ("f") format specifier
- The full date long time ("F") format specifier
- The general date short time ("g") format specifier
- The general date long time ("G") format specifier
- The round-trip ("O", "o") format specifier
- The RFC1123 ("R", "r") format specifier
- The sortable ("s") format specifier
- The universal sortable ("u") format specifier
- The universal full ("U") format specifier

The full date short time ("f") format specifier

The "f" standard format specifier represents a combination of the long date ("D") and short time ("t") patterns, separated by a space.

The result string is affected by the formatting information of a specific [DateTimeFormatInfo](#) object. The following table lists the [DateTimeFormatInfo](#) object properties that may control the formatting of the returned string. The custom format specifier returned by the [DateTimeFormatInfo.LongDatePattern](#) and [DateTimeFormatInfo.ShortTimePattern](#) properties of some cultures may not make use of all properties.

| Property | Description |
|----------------------------------|--|
| LongDatePattern | Defines the format of the date component of the result string. |
| ShortTimePattern | Defines the format of the time component of the result string. |
| DayNames | Defines the localized day names that can appear in the result string. |
| MonthNames | Defines the localized month names that can appear in the result string. |
| TimeSeparator | Defines the string that separates the hour, minute, and second components of a time. |
| AMDesignator | Defines the string that indicates times from midnight to before noon in a 12-hour clock. |
| PMDesignator | Defines the string that indicates times from noon to before midnight in a 12-hour clock. |

The following example uses the "f" format specifier to display a date and time value.

C#

```

DateTime date1 = new DateTime(2008, 4, 10, 6, 30, 0);
Console.WriteLine(date1.ToString("F",
                               CultureInfo.CreateSpecificCulture("en-US")));
// Displays Thursday, April 10, 2008 6:30 AM
Console.WriteLine(date1.ToString("F",
                               CultureInfo.CreateSpecificCulture("fr-FR")));
// Displays jeudi 10 avril 2008 06:30

```

[Back to table](#)

The full date long time ("F") format specifier

The "F" standard format specifier represents a custom date and time format string that is defined by the current [DateTimeFormatInfo.FullDateTimePattern](#) property. For example, the custom format string for the invariant culture is "ddd, dd MMMM yyyy HH:mm:ss".

The following table lists the [DateTimeFormatInfo](#) object properties that may control the formatting of the returned string. The custom format specifier that is returned by the [FullDateTimePattern](#) property of some cultures may not make use of all properties.

| Property | Description |
|-------------------------------------|--|
| FullDateTimePattern | Defines the overall format of the result string. |
| DayNames | Defines the localized day names that can appear in the result string. |
| MonthNames | Defines the localized month names that can appear in the result string. |
| TimeSeparator | Defines the string that separates the hour, minute, and second components of a time. |
| AMDesignator | Defines the string that indicates times from midnight to before noon in a 12-hour clock. |
| PMDesignator | Defines the string that indicates times from noon to before midnight in a 12-hour clock. |

The following example uses the "F" format specifier to display a date and time value.

C#

```

DateTime date1 = new DateTime(2008, 4, 10, 6, 30, 0);
Console.WriteLine(date1.ToString("F",
                               CultureInfo.CreateSpecificCulture("en-US")));
// Displays Thursday, April 10, 2008 6:30:00 AM
Console.WriteLine(date1.ToString("F",

```

```
CultureInfo.CreateSpecificCulture("fr-FR")));
// Displays jeudi 10 avril 2008 06:30:00
```

[Back to table](#)

The general date short time ("g") format specifier

The "g" standard format specifier represents a combination of the short date ("d") and short time ("t") patterns, separated by a space.

The result string is affected by the formatting information of a specific [DateTimeFormatInfo](#) object. The following table lists the [DateTimeFormatInfo](#) object properties that may control the formatting of the returned string. The custom format specifier that is returned by the [DateTimeFormatInfo.ShortDatePattern](#) and [DateTimeFormatInfo.ShortTimePattern](#) properties of some cultures may not make use of all properties.

| Property | Description |
|----------------------------------|--|
| ShortDatePattern | Defines the format of the date component of the result string. |
| ShortTimePattern | Defines the format of the time component of the result string. |
| DateSeparator | Defines the string that separates the year, month, and day components of a date. |
| TimeSeparator | Defines the string that separates the hour, minute, and second components of a time. |
| AMDesignator | Defines the string that indicates times from midnight to before noon in a 12-hour clock. |
| PMDesignator | Defines the string that indicates times from noon to before midnight in a 12-hour clock. |

The following example uses the "g" format specifier to display a date and time value.

C#

```
DateTime date1 = new DateTime(2008, 4, 10, 6, 30, 0);
Console.WriteLine(date1.ToString("g",
    DateTimeFormatInfo.InvariantInfo));
// Displays 04/10/2008 06:30
Console.WriteLine(date1.ToString("g",
    CultureInfo.CreateSpecificCulture("en-us")));
// Displays 4/10/2008 6:30 AM
Console.WriteLine(date1.ToString("g",
```

```
CultureInfo.CreateSpecificCulture("fr-BE")));
// Displays 10/04/2008 6:30
```

[Back to table](#)

The general date long time ("G") format specifier

The "G" standard format specifier represents a combination of the short date ("d") and long time ("T") patterns, separated by a space.

The result string is affected by the formatting information of a specific [DateTimeFormatInfo](#) object. The following table lists the [DateTimeFormatInfo](#) object properties that may control the formatting of the returned string. The custom format specifier that is returned by the [DateTimeFormatInfo.ShortDatePattern](#) and [DateTimeFormatInfo.LongTimePattern](#) properties of some cultures may not make use of all properties.

| Property | Description |
|----------------------------------|--|
| ShortDatePattern | Defines the format of the date component of the result string. |
| LongTimePattern | Defines the format of the time component of the result string. |
| DateSeparator | Defines the string that separates the year, month, and day components of a date. |
| TimeSeparator | Defines the string that separates the hour, minute, and second components of a time. |
| AMDesignator | Defines the string that indicates times from midnight to before noon in a 12-hour clock. |
| PMDesignator | Defines the string that indicates times from noon to before midnight in a 12-hour clock. |

The following example uses the "G" format specifier to display a date and time value.

C#

```
DateTime date1 = new DateTime(2008, 4, 10, 6, 30, 0);
Console.WriteLine(date1.ToString("G",
    DateTimeFormatInfo.InvariantInfo));
// Displays 04/10/2008 06:30:00
Console.WriteLine(date1.ToString("G",
    CultureInfo.CreateSpecificCulture("en-us")));
// Displays 4/10/2008 6:30:00 AM
Console.WriteLine(date1.ToString("G",
```

```
CultureInfo.CreateSpecificCulture("nl-BE")));  
// Displays 10/04/2008 6:30:00
```

[Back to table](#)

The round-trip ("O", "o") format specifier

The "O" or "o" standard format specifier represents a custom date and time format string using a pattern that preserves time zone information and emits a result string that complies with ISO 8601. For [DateTime](#) values, this format specifier is designed to preserve date and time values along with the [DateTime.Kind](#) property in text. The formatted string can be parsed back by using the [DateTime.Parse\(String\)](#), [IFormatProvider](#), [DateTimeStyles](#)) or [DateTime.ParseExact](#) method if the [styles](#) parameter is set to [DateTimeStyles.RoundtripKind](#).

The "O" or "o" standard format specifier corresponds to the "yyyy'-'MM'-'dd'T'HH':'mm':'ss'.fffffffK" custom format string for [DateTime](#) values and to the "yyyy'-'MM'-'dd'T'HH':'mm':'ss'.fffffffzzz" custom format string for [DateTimeOffset](#) values. In this string, the pairs of single quotation marks that delimit individual characters, such as the hyphens, the colons, and the letter "T", indicate that the individual character is a literal that cannot be changed. The apostrophes do not appear in the output string.

The "O" or "o" standard format specifier (and the "yyyy'-'MM'-'dd'T'HH':'mm':'ss'.fffffffK" custom format string) takes advantage of the three ways that ISO 8601 represents time zone information to preserve the [Kind](#) property of [DateTime](#) values:

- The time zone component of [DateTimeKind.Local](#) date and time values is an offset from UTC (for example, +01:00, -07:00). All [DateTimeOffset](#) values are also represented in this format.
- The time zone component of [DateTimeKind.Utc](#) date and time values uses "Z" (which stands for zero offset) to represent UTC.
- [DateTimeKind.Unspecified](#) date and time values have no time zone information.

Because the "O" or "o" standard format specifier conforms to an international standard, the formatting or parsing operation that uses the specifier always uses the invariant culture and the Gregorian calendar.

Strings that are passed to the [Parse](#), [TryParse](#), [ParseExact](#), and [TryParseExact](#) methods of [DateTime](#) and [DateTimeOffset](#) can be parsed by using the "O" or "o" format specifier if they are in one of these formats. In the case of [DateTime](#) objects, the parsing overload that you call should also include a [styles](#) parameter with a value of

[DateTimeStyles.RoundtripKind](#). Note that if you call a parsing method with the custom format string that corresponds to the "O" or "o" format specifier, you won't get the same results as "O" or "o". This is because parsing methods that use a custom format string can't parse the string representation of date and time values that lack a time zone component or use "Z" to indicate UTC.

The following example uses the "o" format specifier to display a series of [DateTime](#) values and a [DateTimeOffset](#) value on a system in the U.S. Pacific Time zone.

C#

```
using System;

public class Example
{
    public static void Main()
    {
        DateTime dat = new DateTime(2009, 6, 15, 13, 45, 30,
                                    DateTimeKind.Unspecified);
        Console.WriteLine("{0} ({1}) --> {0:0}", dat, dat.Kind);

        DateTime uDat = new DateTime(2009, 6, 15, 13, 45, 30,
                                    DateTimeKind.Utc);
        Console.WriteLine("{0} ({1}) --> {0:0}", uDat, uDat.Kind);

        DateTime lDat = new DateTime(2009, 6, 15, 13, 45, 30,
                                    DateTimeKind.Local);
        Console.WriteLine("{0} ({1}) --> {0:0}\n", lDat, lDat.Kind);

        DateTimeOffset dto = new DateTimeOffset(lDat);
        Console.WriteLine("{0} --> {0:0}", dto);
    }
}

// The example displays the following output:
//   6/15/2009 1:45:30 PM (Unspecified) --> 2009-06-15T13:45:30.0000000
//   6/15/2009 1:45:30 PM (Utc) --> 2009-06-15T13:45:30.0000000Z
//   6/15/2009 1:45:30 PM (Local) --> 2009-06-15T13:45:30.0000000-07:00
//
//   6/15/2009 1:45:30 PM -07:00 --> 2009-06-15T13:45:30.0000000-07:00
```

The following example uses the "o" format specifier to create a formatted string, and then restores the original date and time value by calling a date and time [Parse](#) method.

C#

```
// Round-trip DateTime values.
DateTime originalDate, newDate;
string dateString;
// Round-trip a local time.
originalDate = DateTime.SpecifyKind(new DateTime(2008, 4, 10, 6, 30, 0),
```

```

DateTimeKind.Local);
dateString = originalDate.ToString("o");
newDate = DateTime.Parse(dateString, null, DateTimeStyles.RoundtripKind);
Console.WriteLine("Round-tripped {0} {1} to {2} {3}.", originalDate,
originalDate.Kind,
                newDate, newDate.Kind);
// Round-trip a UTC time.
originalDate = DateTime.SpecifyKind(new DateTime(2008, 4, 12, 9, 30, 0),
DateTimeKind.Utc);
dateString = originalDate.ToString("o");
newDate = DateTime.Parse(dateString, null, DateTimeStyles.RoundtripKind);
Console.WriteLine("Round-tripped {0} {1} to {2} {3}.", originalDate,
originalDate.Kind,
                newDate, newDate.Kind);
// Round-trip time in an unspecified time zone.
originalDate = DateTime.SpecifyKind(new DateTime(2008, 4, 13, 12, 30, 0),
DateTimeKind.Unspecified);
dateString = originalDate.ToString("o");
newDate = DateTime.Parse(dateString, null, DateTimeStyles.RoundtripKind);
Console.WriteLine("Round-tripped {0} {1} to {2} {3}.", originalDate,
originalDate.Kind,
                newDate, newDate.Kind);

// Round-trip a DateTimeOffset value.
DateTimeOffset originalDTO = new DateTimeOffset(2008, 4, 12, 9, 30, 0, new
TimeSpan(-8, 0, 0));
dateString = originalDTO.ToString("o");
DateTimeOffset newDTO = DateTimeOffset.Parse(dateString, null,
DateTimeStyles.RoundtripKind);
Console.WriteLine("Round-tripped {0} to {1}.", originalDTO, newDTO);
// The example displays the following output:
//    Round-tripped 4/10/2008 6:30:00 AM Local to 4/10/2008 6:30:00 AM
Local.
//    Round-tripped 4/12/2008 9:30:00 AM Utc to 4/12/2008 9:30:00 AM Utc.
//    Round-tripped 4/13/2008 12:30:00 PM Unspecified to 4/13/2008 12:30:00
PM Unspecified.
//    Round-tripped 4/12/2008 9:30:00 AM -08:00 to 4/12/2008 9:30:00 AM
-08:00.

```

[Back to table](#)

The RFC1123 ("R", "r") format specifier

The "R" or "r" standard format specifier represents a custom date and time format string that's defined by the [DateTimeFormatInfo.RFC1123Pattern](#) property. The pattern reflects a defined standard, and the property is read-only. Therefore, it is always the same, regardless of the culture used or the format provider supplied. The custom format string is "ddd, dd MMM yyyy HH':'mm':'ss 'GMT'". When this standard format specifier is used, the formatting or parsing operation always uses the invariant culture.

The result string is affected by the following properties of the [DateTimeFormatInfo](#) object returned by the [DateTimeFormatInfo.InvariantInfo](#) property that represents the invariant culture.

| Property | Description |
|-----------------------|---|
| RFC1123Pattern | Defines the format of the result string. |
| AbbreviatedDayNames | Defines the abbreviated day names that can appear in the result string. |
| AbbreviatedMonthNames | Defines the abbreviated month names that can appear in the result string. |

Although the RFC 1123 standard expresses a time as Coordinated Universal Time (UTC), the formatting operation does not modify the value of the [DateTime](#) object that's being formatted. Therefore, you must convert the [DateTime](#) value to UTC by calling the [DateTime.ToUniversalTime](#) method before you perform the formatting operation. In contrast, [DateTimeOffset](#) values perform this conversion automatically; there's no need to call the [DateTimeOffset.ToUniversalTime](#) method before the formatting operation.

The following example uses the "r" format specifier to display a [DateTime](#) and a [DateTimeOffset](#) value on a system in the U.S. Pacific Time zone.

C#

```
DateTime date1 = new DateTime(2008, 4, 10, 6, 30, 0);
DateTimeOffset dateOffset = new DateTimeOffset(date1,
    TimeZoneInfo.Local.GetUtcOffset(date1));
Console.WriteLine(date1.ToUniversalTime().ToString("r"));
// Displays Thu, 10 Apr 2008 13:30:00 GMT
Console.WriteLine(dateOffset.ToUniversalTime().ToString("r"));
// Displays Thu, 10 Apr 2008 13:30:00 GMT
```

[Back to table](#)

The sortable ("s") format specifier

The "s" standard format specifier represents a custom date and time format string that is defined by the [DateTimeFormatInfo.SortableDateTimePattern](#) property. The pattern reflects a defined standard (ISO 8601), and the property is read-only. Therefore, it is always the same, regardless of the culture used or the format provider supplied. The custom format string is "yyyy'-'MM'-'dd'T'HH':'mm':'ss".

The purpose of the "s" format specifier is to produce result strings that sort consistently in ascending or descending order based on date and time values. As a result, although the "s" standard format specifier represents a date and time value in a consistent format, the formatting operation does not modify the value of the date and time object that is being formatted to reflect its [DateTime.Kind](#) property or its [DateTimeOffset.Offset](#) value. For example, the result strings produced by formatting the date and time values 2014-11-15T18:32:17+00:00 and 2014-11-15T18:32:17+08:00 are identical.

When this standard format specifier is used, the formatting or parsing operation always uses the invariant culture.

The following example uses the "s" format specifier to display a [DateTime](#) and a [DateTimeOffset](#) value on a system in the U.S. Pacific Time zone.

```
C#
```

```
DateTime date1 = new DateTime(2008, 4, 10, 6, 30, 0);
Console.WriteLine(date1.ToString("s"));
// Displays 2008-04-10T06:30:00
```

[Back to table](#)

The universal sortable ("u") format specifier

The "u" standard format specifier represents a custom date and time format string that is defined by the [DateTimeFormatInfo.UniversalSortableDateTimePattern](#) property. The pattern reflects a defined standard, and the property is read-only. Therefore, it is always the same, regardless of the culture used or the format provider supplied. The custom format string is "yyyy'-'MM'-'dd HH':'mm':'ss'Z"". When this standard format specifier is used, the formatting or parsing operation always uses the invariant culture.

Although the result string should express a time as Coordinated Universal Time (UTC), no conversion of the original [DateTime](#) value is performed during the formatting operation. Therefore, you must convert a [DateTime](#) value to UTC by calling the [DateTime.ToUniversalTime](#) method before formatting it. In contrast, [DateTimeOffset](#) values perform this conversion automatically; there is no need to call the [DateTimeOffset.ToUniversalTime](#) method before the formatting operation.

The following example uses the "u" format specifier to display a date and time value.

```
C#
```

```
DateTime date1 = new DateTime(2008, 4, 10, 6, 30, 0);
Console.WriteLine(date1.ToUniversalTime().ToString("u"));
```

```
// Displays 2008-04-10 13:30:00Z
```

[Back to table](#)

The universal full ("U") format specifier

The "U" standard format specifier represents a custom date and time format string that is defined by a specified culture's [DateTimeFormatInfo.FullDateTimePattern](#) property.

The pattern is the same as the "F" pattern. However, the [DateTime](#) value is automatically converted to UTC before it is formatted.

The following table lists the [DateTimeFormatInfo](#) object properties that may control the formatting of the returned string. The custom format specifier that is returned by the [FullDateTimePattern](#) property of some cultures may not make use of all properties.

| Property | Description |
|-------------------------------------|--|
| FullDateTimePattern | Defines the overall format of the result string. |
| DayNames | Defines the localized day names that can appear in the result string. |
| MonthNames | Defines the localized month names that can appear in the result string. |
| TimeSeparator | Defines the string that separates the hour, minute, and second components of a time. |
| AMDesignator | Defines the string that indicates times from midnight to before noon in a 12-hour clock. |
| PMDesignator | Defines the string that indicates times from noon to before midnight in a 12-hour clock. |

The "U" format specifier is not supported by the [DateTimeOffset](#) type and throws a [FormatException](#) if it is used to format a [DateTimeOffset](#) value.

The following example uses the "U" format specifier to display a date and time value.

C#

```
DateTime date1 = new DateTime(2008, 4, 10, 6, 30, 0);
Console.WriteLine(date1.ToString("U",
                           CultureInfo.CreateSpecificCulture("en-US")));
// Displays Thursday, April 10, 2008 1:30:00 PM
Console.WriteLine(date1.ToString("U",
                           CultureInfo.CreateSpecificCulture("sv-FI")));
// Displays den 10 april 2008 13:30:00
```

Time formats

This group includes the following formats:

- The short time ("t") format specifier
- The long time ("T") format specifier

The short time ("t") format specifier

The "t" standard format specifier represents a custom date and time format string that is defined by the current [DateTimeFormatInfo.ShortTimePattern](#) property. For example, the custom format string for the invariant culture is "HH:mm".

The result string is affected by the formatting information of a specific [DateTimeFormatInfo](#) object. The following table lists the [DateTimeFormatInfo](#) object properties that may control the formatting of the returned string. The custom format specifier that is returned by the [DateTimeFormatInfo.ShortTimePattern](#) property of some cultures may not make use of all properties.

| Property | Description |
|----------------------------------|--|
| ShortTimePattern | Defines the format of the time component of the result string. |
| TimeSeparator | Defines the string that separates the hour, minute, and second components of a time. |
| AMDesignator | Defines the string that indicates times from midnight to before noon in a 12-hour clock. |
| PMDesignator | Defines the string that indicates times from noon to before midnight in a 12-hour clock. |

The following example uses the "t" format specifier to display a date and time value.

C#

```
DateTime date1 = new DateTime(2008, 4, 10, 6, 30, 0);
Console.WriteLine(date1.ToString("t",
    CultureInfo.CreateSpecificCulture("en-us")));
// Displays 6:30 AM
Console.WriteLine(date1.ToString("t",
    CultureInfo.CreateSpecificCulture("es-ES")));
// Displays 6:30
```

[Back to table](#)

The long time ("T") format specifier

The "T" standard format specifier represents a custom date and time format string that is defined by a specific culture's [DateTimeFormatInfo.LongTimePattern](#) property. For example, the custom format string for the invariant culture is "HH:mm:ss".

The following table lists the [DateTimeFormatInfo](#) object properties that may control the formatting of the returned string. The custom format specifier that is returned by the [DateTimeFormatInfo.LongTimePattern](#) property of some cultures may not make use of all properties.

| Property | Description |
|---------------------------------|--|
| LongTimePattern | Defines the format of the time component of the result string. |
| TimeSeparator | Defines the string that separates the hour, minute, and second components of a time. |
| AMDesignator | Defines the string that indicates times from midnight to before noon in a 12-hour clock. |
| PMDesignator | Defines the string that indicates times from noon to before midnight in a 12-hour clock. |

The following example uses the "T" format specifier to display a date and time value.

C#

```
DateTime date1 = new DateTime(2008, 4, 10, 6, 30, 0);
Console.WriteLine(date1.ToString("T",
    CultureInfo.CreateSpecificCulture("en-US")));
// Displays 6:30:00 AM
Console.WriteLine(date1.ToString("T",
    CultureInfo.CreateSpecificCulture("es-ES")));
// Displays 6:30:00
```

[Back to table](#)

Partial date formats

This group includes the following formats:

- [The month \("M", "m"\) format specifier](#)
- [The year month \("Y", "y"\) format specifier](#)

The month ("M", "m") format specifier

The "M" or "m" standard format specifier represents a custom date and time format string that is defined by the current [DateTimeFormatInfo.MonthDayPattern](#) property. For example, the custom format string for the invariant culture is "MMMM dd".

The following table lists the [DateTimeFormatInfo](#) object properties that control the formatting of the returned string.

| Property | Description |
|---------------------------------|---|
| MonthDayPattern | Defines the overall format of the result string. |
| MonthNames | Defines the localized month names that can appear in the result string. |

The following example uses the "m" format specifier to display a date and time value.

C#

```
DateTime date1 = new DateTime(2008, 4, 10, 6, 30, 0);
Console.WriteLine(date1.ToString("m",
    CultureInfo.CreateSpecificCulture("en-us")));
// Displays April 10
Console.WriteLine(date1.ToString("m",
    CultureInfo.CreateSpecificCulture("ms-MY")));
// Displays 10 April
```

[Back to table](#)

The year month ("Y", "y") format specifier

The "Y" or "y" standard format specifier represents a custom date and time format string that is defined by the [DateTimeFormatInfo.YearMonthPattern](#) property of a specified culture. For example, the custom format string for the invariant culture is "yyyy MMMM".

The following table lists the [DateTimeFormatInfo](#) object properties that control the formatting of the returned string.

| Property | Description |
|----------------------------------|---|
| YearMonthPattern | Defines the overall format of the result string. |
| MonthNames | Defines the localized month names that can appear in the result string. |

The following example uses the "y" format specifier to display a date and time value.

C#

```
DateTime date1 = new DateTime(2008, 4, 10, 6, 30, 0);
Console.WriteLine(date1.ToString("Y",
    CultureInfo.CreateSpecificCulture("en-US")));
// Displays April, 2008
Console.WriteLine(date1.ToString("y",
    CultureInfo.CreateSpecificCulture("af-ZA")));
// Displays April 2008
```

[Back to table](#)

Control Panel settings

In Windows, the settings in the **Regional and Language Options** item in Control Panel influence the result string produced by a formatting operation. These settings are used to initialize the [DateTimeFormatInfo](#) object associated with the current culture, which provides values used to govern formatting. Computers that use different settings generate different result strings.

In addition, if you use the [CultureInfo\(String\)](#) constructor to instantiate a new [CultureInfo](#) object that represents the same culture as the current system culture, any customizations established by the **Regional and Language Options** item in Control Panel will be applied to the new [CultureInfo](#) object. You can use the [CultureInfo\(String, Boolean\)](#) constructor to create a [CultureInfo](#) object that does not reflect a system's customizations.

DateTimeFormatInfo properties

Formatting is influenced by properties of the current [DateTimeFormatInfo](#) object, which is provided implicitly by the current culture or explicitly by the [IFormatProvider](#) parameter of the method that invokes formatting. For the [IFormatProvider](#) parameter, your application should specify a [CultureInfo](#) object, which represents a culture, or a [DateTimeFormatInfo](#) object, which represents a particular culture's date and time formatting conventions. Many of the standard date and time format specifiers are aliases for formatting patterns defined by properties of the current [DateTimeFormatInfo](#) object. Your application can change the result produced by some standard date and time format specifiers by changing the corresponding date and time format patterns of the corresponding [DateTimeFormatInfo](#) property.

See also

- [System.DateTime](#)
- [System.DateTimeOffset](#)
- [Formatting Types](#)
- [Custom Date and Time Format Strings](#)
- [Sample: .NET Core WinForms Formatting Utility \(C#\)](#)
- [Sample: .NET Core WinForms Formatting Utility \(Visual Basic\)](#)

 [Collaborate with us on GitHub](#)

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

Custom date and time format strings

Article • 12/04/2022

A date and time format string defines the text representation of a [DateTime](#) or [DateTimeOffset](#) value that results from a formatting operation. It can also define the representation of a date and time value that is required in a parsing operation in order to successfully convert the string to a date and time. A custom format string consists of one or more custom date and time format specifiers. Any string that is not a [standard date and time format string](#) is interpreted as a custom date and time format string.

Tip

You can download the [Formatting Utility](#), a .NET Core Windows Forms application that lets you apply format strings to either numeric or date and time values and displays the result string. Source code is available for [C#](#) and [Visual Basic](#).

Custom date and time format strings can be used with both [DateTime](#) and [DateTimeOffset](#) values.

Note

Some of the C# examples in this article run in the [Try.NET](#) inline code runner and playground. Select the **Run** button to run an example in an interactive window. Once you execute the code, you can modify it and run the modified code by selecting **Run** again. The modified code either runs in the interactive window or, if compilation fails, the interactive window displays all C# compiler error messages.

The **local time zone** of the [Try.NET](#) inline code runner and playground is Coordinated Universal Time, or UTC. This may affect the behavior and the output of examples that illustrate the [DateTime](#), [DateTimeOffset](#), and [TimeZoneInfo](#) types and their members.

In formatting operations, custom date and time format strings can be used either with the `ToString` method of a date and time instance or with a method that supports composite formatting. The following example illustrates both uses.

C#

```
DateTime thisDate1 = new DateTime(2011, 6, 10);
Console.WriteLine("Today is " + thisDate1.ToString("MMMM dd, yyyy") + ".");
```

```

DateTimeOffset thisDate2 = new DateTimeOffset(2011, 6, 10, 15, 24, 16,
                                              TimeSpan.Zero);
Console.WriteLine("The current date and time: {0:MM/dd/yy H:mm:ss zzz}",
                  thisDate2);
// The example displays the following output:
// Today is June 10, 2011.
// The current date and time: 06/10/11 15:24:16 +00:00

```

In parsing operations, custom date and time format strings can be used with the [DateTime.ParseExact](#), [DateTime.TryParseExact](#), [DateTimeOffset.ParseExact](#), and [DateTimeOffset.TryParseExact](#) methods. These methods require that an input string conforms exactly to a particular pattern for the parse operation to succeed. The following example illustrates a call to the [DateTimeOffset.ParseExact\(String, String, IFormatProvider\)](#) method to parse a date that must include a day, a month, and a two-digit year.

C#

```

using System;
using System.Globalization;

public class Example1
{
    public static void Main()
    {
        string[] dateValues = { "30-12-2011", "12-30-2011",
                               "30-12-11", "12-30-11" };
        string pattern = "MM-dd-yy";
        DateTime parsedDate;

        foreach (var dateValue in dateValues)
        {
            if (DateTime.TryParseExact(dateValue, pattern, null,
                                      DateTimeStyles.None, out parsedDate))
                Console.WriteLine("Converted '{0}' to {1:d}.",
                                  dateValue, parsedDate);
            else
                Console.WriteLine("Unable to convert '{0}' to a date and
time.",
                                  dateValue);
        }
    }
}
// The example displays the following output:
// Unable to convert '30-12-2011' to a date and time.
// Unable to convert '12-30-2011' to a date and time.
// Unable to convert '30-12-11' to a date and time.
// Converted '12-30-11' to 12/30/2011.

```

The following table describes the custom date and time format specifiers and displays a result string produced by each format specifier. By default, result strings reflect the formatting conventions of the en-US culture. If a particular format specifier produces a localized result string, the example also notes the culture to which the result string applies. For more information about using custom date and time format strings, see the [Notes](#) section.

| Format specifier | Description | Examples |
|-------------------------|--|---|
| "d" | The day of the month, from 1 to 31. More information: The "d" Custom Format Specifier . | 2009-06-01T13:45:30 -> 1 2009-06-15T13:45:30 -> 15 |
| "dd" | The day of the month, from 01 to 31. More information: The "dd" Custom Format Specifier . | 2009-06-01T13:45:30 -> 01 2009-06-15T13:45:30 -> 15 |
| "ddd" | The abbreviated name of the day of the week. More information: The "ddd" Custom Format Specifier . | 2009-06-15T13:45:30 -> Mon (en-US) 2009-06-15T13:45:30 -> Пн (ru-RU) 2009-06-15T13:45:30 -> lun. (fr-FR) |
| "dddd" | The full name of the day of the week. More information: The "dddd" Custom Format Specifier . | 2009-06-15T13:45:30 -> Monday (en-US) 2009-06-15T13:45:30 -> понедельник (ru-RU) 2009-06-15T13:45:30 -> lundi (fr-FR) |
| "f" | The tenths of a second in a date and time value. More information: The "f" Custom Format Specifier . | 2009-06-15T13:45:30.6170000 -> 6 2009-06-15T13:45:30.05 -> 0 |
| "ff" | The hundredths of a second in a date and time value. More information: The "ff" Custom Format Specifier . | 2009-06-15T13:45:30.6170000 -> 61 2009-06-15T13:45:30.0050000 -> 00 |
| "fff" | The milliseconds in a date and time value. | 6/15/2009 13:45:30.617 -> 617 6/15/2009 13:45:30.0005 -> 000 |

| Format specifier | Description | Examples |
|------------------|---|--|
| | More information: The "fff" Custom Format Specifier . | |
| 'ffff' | <p>The ten thousandths of a second in a date and time value.</p> <p>More information: The "ffff" Custom Format Specifier.</p> | <p>2009-06-15T13:45:30.6175000 -> 6175 2009-06-15T13:45:30.0000500 -> 0000</p> |
| 'fffff' | <p>The hundred thousandths of a second in a date and time value.</p> <p>More information: The "fffff" Custom Format Specifier.</p> | <p>2009-06-15T13:45:30.6175400 -> 61754 6/15/2009 13:45:30.000005 -> 00000</p> |
| 'ffffff' | <p>The millionths of a second in a date and time value.</p> <p>More information: The "ffffff" Custom Format Specifier.</p> | <p>2009-06-15T13:45:30.6175420 -> 617542 2009-06-15T13:45:30.0000005 -> 000000</p> |
| 'fffffff' | <p>The ten millionths of a second in a date and time value.</p> <p>More information: The "fffffff" Custom Format Specifier.</p> | <p>2009-06-15T13:45:30.6175425 -> 6175425 2009-06-15T13:45:30.0001150 -> 0001150</p> |
| 'F' | <p>If non-zero, the tenths of a second in a date and time value.</p> <p>More information: The "F" Custom Format Specifier.</p> | <p>2009-06-15T13:45:30.6170000 -> 6 2009-06-15T13:45:30.0500000 -> (no output)</p> |
| 'FF' | <p>If non-zero, the hundredths of a second in a date and time value.</p> <p>More information: The "FF" Custom Format Specifier.</p> | <p>2009-06-15T13:45:30.6170000 -> 61 2009-06-15T13:45:30.0050000 -> (no output)</p> |
| 'FFF' | <p>If non-zero, the milliseconds in a date and time value.</p> <p>More information: The "FFF" Custom Format Specifier.</p> | <p>2009-06-15T13:45:30.6170000 -> 617 2009-06-15T13:45:30.0005000 -> (no output)</p> |
| 'FFFF' | <p>If non-zero, the ten thousandths of a second in a date and time value.</p> | <p>2009-06-15T13:45:30.5275000 -> 5275</p> |

| Format specifier | Description | Examples |
|-------------------------|--|--|
| | More information: The "FFFF" Custom Format Specifier. | 2009-06-15T13:45:30.0000500 -> (no output) |
| "FFFF" | If non-zero, the hundred thousandths of a second in a date and time value. | 2009-06-15T13:45:30.6175400 -> 61754 |
| | More information: The "FFFF" Custom Format Specifier. | 2009-06-15T13:45:30.0000050 -> (no output) |
| "FFFFFF" | If non-zero, the millionths of a second in a date and time value. | 2009-06-15T13:45:30.6175420 -> 617542 |
| | More information: The "FFFFFF" Custom Format Specifier. | 2009-06-15T13:45:30.0000005 -> (no output) |
| "FFFFFFF" | If non-zero, the ten millionths of a second in a date and time value. | 2009-06-15T13:45:30.6175425 -> 6175425 |
| | More information: The "FFFFFFF" Custom Format Specifier. | 2009-06-15T13:45:30.0001150 -> 000115 |
| "g", "gg" | The period or era. | 2009-06-15T13:45:30.6170000 -> A.D. |
| | More information: The "g" or "gg" Custom Format Specifier. | |
| "h" | The hour, using a 12-hour clock from 1 to 12. | 2009-06-15T01:45:30 -> 1 |
| | More information: The "h" Custom Format Specifier. | 2009-06-15T13:45:30 -> 1 |
| "hh" | The hour, using a 12-hour clock from 01 to 12. | 2009-06-15T01:45:30 -> 01 |
| | More information: The "hh" Custom Format Specifier. | 2009-06-15T13:45:30 -> 01 |
| "H" | The hour, using a 24-hour clock from 0 to 23. | 2009-06-15T01:45:30 -> 1 |
| | More information: The "H" Custom Format Specifier. | 2009-06-15T13:45:30 -> 13 |
| "HH" | The hour, using a 24-hour clock from 00 to 23. | 2009-06-15T01:45:30 -> 01 |
| | | 2009-06-15T13:45:30 -> 13 |

| Format specifier | Description | Examples |
|-------------------------|---|--|
| | More information: The "HH" Custom Format Specifier. | |
| "K" | <p>Time zone information.</p> <p>More information: The "K" Custom Format Specifier.</p> | <p>With DateTime values:</p> <p>2009-06-15T13:45:30, Kind Unspecified -></p> <p>2009-06-15T13:45:30, Kind Utc -> Z</p> <p>2009-06-15T13:45:30, Kind Local -> -07:00 (depends on local computer settings)</p> <p>With DateTimeOffset values:</p> <p>2009-06-15T01:45:30-07:00 --> -07:00</p> <p>2009-06-15T08:45:30+00:00 --> +00:00</p> |
| "m" | <p>The minute, from 0 to 59.</p> <p>More information: The "m" Custom Format Specifier.</p> | 2009-06-15T01:09:30 -> 9 |
| "mm" | <p>The minute, from 00 to 59.</p> <p>More information: The "mm" Custom Format Specifier.</p> | <p>2009-06-15T01:09:30 -> 09</p> <p>2009-06-15T01:45:30 -> 45</p> |
| "M" | <p>The month, from 1 to 12.</p> <p>More information: The "M" Custom Format Specifier.</p> | 2009-06-15T13:45:30 -> 6 |
| "MM" | <p>The month, from 01 to 12.</p> <p>More information: The "MM" Custom Format Specifier.</p> | 2009-06-15T13:45:30 -> 06 |
| "MMM" | <p>The abbreviated name of the month.</p> <p>More information: The "MMM" Custom Format Specifier.</p> | <p>2009-06-15T13:45:30 -> Jun (en-US)</p> <p>2009-06-15T13:45:30 -> juin (fr-FR)</p> <p>2009-06-15T13:45:30 -> Jun (zu-ZA)</p> |
| "MMMM" | The full name of the month. | 2009-06-15T13:45:30 -> June (en-US) |

| Format specifier | Description | Examples |
|-------------------------|--|---|
| | More information: The "MMMM" Custom Format Specifier. | 2009-06-15T13:45:30 -> juni (da-DK) 2009-06-15T13:45:30 -> uJuni (zu-ZA) |
| "s" | The second, from 0 to 59. More information: The "s" Custom Format Specifier. | 2009-06-15T13:45:09 -> 9 |
| "ss" | The second, from 00 to 59. More information: The "ss" Custom Format Specifier. | 2009-06-15T13:45:09 -> 09 |
| "t" | The first character of the AM/PM designator. More information: The "t" Custom Format Specifier. | 2009-06-15T13:45:30 -> P (en-US) 2009-06-15T13:45:30 -> 午 (ja-JP) 2009-06-15T13:45:30 -> (fr-FR) |
| "tt" | The AM/PM designator. More information: The "tt" Custom Format Specifier. | 2009-06-15T13:45:30 -> PM (en-US) 2009-06-15T13:45:30 -> 午後 (ja-JP) 2009-06-15T13:45:30 -> (fr-FR) |
| "y" | The year, from 0 to 99. More information: The "y" Custom Format Specifier. | 0001-01-01T00:00:00 -> 1 0900-01-01T00:00:00 -> 0 1900-01-01T00:00:00 -> 0 2009-06-15T13:45:30 -> 9 2019-06-15T13:45:30 -> 19 |
| "yy" | The year, from 00 to 99. More information: The "yy" Custom Format Specifier. | 0001-01-01T00:00:00 -> 01 0900-01-01T00:00:00 -> 00 1900-01-01T00:00:00 -> 00 2019-06-15T13:45:30 -> 19 |
| "yyy" | The year, with a minimum of three digits. | 0001-01-01T00:00:00 -> 001 0900-01-01T00:00:00 -> 900 |

| Format specifier | Description | Examples |
|---|---|--|
| | More information: The "yyy" Custom Format Specifier. | 1900-01-01T00:00:00 -> 1900 2009-06-15T13:45:30 -> 2009 |
| "yyyy" | The year as a four-digit number. More information: The "yyyy" Custom Format Specifier. | 0001-01-01T00:00:00 -> 0001 0900-01-01T00:00:00 -> 0900 1900-01-01T00:00:00 -> 1900 2009-06-15T13:45:30 -> 2009 |
| "yyyyy" | The year as a five-digit number. More information: The "yyyyy" Custom Format Specifier. | 0001-01-01T00:00:00 -> 00001 2009-06-15T13:45:30 -> 02009 |
| "z" | Hours offset from UTC, with no leading zeros. More information: The "z" Custom Format Specifier. | 2009-06-15T13:45:30-07:00 -> -7 |
| "zz" | Hours offset from UTC, with a leading zero for a single-digit value. More information: The "zz" Custom Format Specifier. | 2009-06-15T13:45:30-07:00 -> -07 |
| "zzz" | Hours and minutes offset from UTC. More information: The "zzz" Custom Format Specifier. | 2009-06-15T13:45:30-07:00 -> -07:00 |
| The time separator. More information: The ":" Custom Format Specifier. | 2009-06-15T13:45:30 -> : (en-US) 2009-06-15T13:45:30 -> . (it-IT) 2009-06-15T13:45:30 -> : (ja-JP) | |
| "/" | The date separator. More Information: The "/" Custom Format Specifier. | 2009-06-15T13:45:30 -> / (en-US) 2009-06-15T13:45:30 -> - (ar-DZ) 2009-06-15T13:45:30 -> . (tr-TR) |
| "string" | Literal string delimiter. | 2009-06-15T13:45:30 ("arr:" h:m t) -> arr: 1:45 P |

| Format specifier | Description | Examples |
|---------------------|--|---|
| 'string' | More information: Character literals . | 2009-06-15T13:45:30 ('arr:' h:m t) -> arr: 1:45 P |
| % | Defines the following character as a custom format specifier. More information: Using Single Custom Format Specifiers . | 2009-06-15T13:45:30 (%h) -> 1 |
| \ | The escape character. More information: Character literals and Using the Escape Character . | 2009-06-15T13:45:30 (h \h) -> 1 h |
| Any other character | The character is copied to the result string unchanged. More information: Character literals . | 2009-06-15T01:45:30 (arr hh:mm t) -> arr 01:45 A |

The following sections provide additional information about each custom date and time format specifier. Unless otherwise noted, each specifier produces an identical string representation regardless of whether it's used with a [DateTime](#) value or a [DateTimeOffset](#) value.

Day "d" format specifier

The "d" custom format specifier

The "d" custom format specifier represents the day of the month as a number from 1 to 31. A single-digit day is formatted without a leading zero.

If the "d" format specifier is used without other custom format specifiers, it's interpreted as the "d" standard date and time format specifier. For more information about using a single format specifier, see [Using Single Custom Format Specifiers](#) later in this article.

The following example includes the "d" custom format specifier in several format strings.

C#

```
DateTime date1 = new DateTime(2008, 8, 29, 19, 27, 15);

Console.WriteLine(date1.ToString("d, M",
    CultureInfo.InvariantCulture));
```

```
// Displays 29, 8

Console.WriteLine(date1.ToString("d MMMM",
    CultureInfo.CreateSpecificCulture("en-US")));
// Displays 29 August
Console.WriteLine(date1.ToString("d MMMM",
    CultureInfo.CreateSpecificCulture("es-MX")));
// Displays 29 agosto
```

[Back to table](#)

The "dd" custom format specifier

The "dd" custom format string represents the day of the month as a number from 01 to 31. A single-digit day is formatted with a leading zero.

The following example includes the "dd" custom format specifier in a custom format string.

C#

```
DateTime date1 = new DateTime(2008, 1, 2, 6, 30, 15);

Console.WriteLine(date1.ToString("dd, MM",
    CultureInfo.InvariantCulture));
// 02, 01
```

[Back to table](#)

The "ddd" custom format specifier

The "ddd" custom format specifier represents the abbreviated name of the day of the week. The localized abbreviated name of the day of the week is retrieved from the [DateTimeFormatInfo.AbbreviatedDayNames](#) property of the current or specified culture.

The following example includes the "ddd" custom format specifier in a custom format string.

C#

```
DateTime date1 = new DateTime(2008, 8, 29, 19, 27, 15);

Console.WriteLine(date1.ToString("ddd d MMM",
    CultureInfo.CreateSpecificCulture("en-US")));
// Displays Fri 29 Aug
Console.WriteLine(date1.ToString("ddd d MMM",
```

```
CultureInfo.CreateSpecificCulture("fr-FR")));
// Displays ven. 29 août
```

[Back to table](#)

The "dddd" custom format specifier

The "dddd" custom format specifier (plus any number of additional "d" specifiers) represents the full name of the day of the week. The localized name of the day of the week is retrieved from the [DateTimeFormatInfo.DayNames](#) property of the current or specified culture.

The following example includes the "dddd" custom format specifier in a custom format string.

C#

```
DateTime date1 = new DateTime(2008, 8, 29, 19, 27, 15);

Console.WriteLine(date1.ToString("dddd dd MMMM",
                               CultureInfo.CreateSpecificCulture("en-US")));
// Displays Friday 29 August
Console.WriteLine(date1.ToString("dddd dd MMMM",
                               CultureInfo.CreateSpecificCulture("it-IT")));
// Displays venerdì 29 agosto
```

[Back to table](#)

Lowercase seconds "f" fraction specifier

The "f" custom format specifier

The "f" custom format specifier represents the most significant digit of the seconds fraction; that is, it represents the tenths of a second in a date and time value.

If the "f" format specifier is used without other format specifiers, it's interpreted as the "f" standard date and time format specifier. For more information about using a single format specifier, see [Using Single Custom Format Specifiers](#) later in this article.

When you use "f" format specifiers as part of a format string supplied to the [ParseExact](#), [TryParseExact](#), [ParseExact](#), or [TryParseExact](#) method, the number of "f" format specifiers indicates the number of most significant digits of the seconds fraction that must be present to successfully parse the string.

The following example includes the "f" custom format specifier in a custom format string.

C#

```
DateTime date1 = new DateTime(2008, 8, 29, 19, 27, 15, 18);
CultureInfo ci = CultureInfo.InvariantCulture;

Console.WriteLine(date1.ToString("hh:mm:ss.f", ci));
// Displays 07:27:15.0
Console.WriteLine(date1.ToString("hh:mm:ss.F", ci));
// Displays 07:27:15
Console.WriteLine(date1.ToString("hh:mm:ss.ff", ci));
// Displays 07:27:15.01
Console.WriteLine(date1.ToString("hh:mm:ss.FF", ci));
// Displays 07:27:15.01
Console.WriteLine(date1.ToString("hh:mm:ss.fff", ci));
// Displays 07:27:15.018
Console.WriteLine(date1.ToString("hh:mm:ss.FFF", ci));
// Displays 07:27:15.018
```

[Back to table](#)

The "ff" custom format specifier

The "ff" custom format specifier represents the two most significant digits of the seconds fraction; that is, it represents the hundredths of a second in a date and time value.

following example includes the "ff" custom format specifier in a custom format string.

C#

```
DateTime date1 = new DateTime(2008, 8, 29, 19, 27, 15, 18);
CultureInfo ci = CultureInfo.InvariantCulture;

Console.WriteLine(date1.ToString("hh:mm:ss.f", ci));
// Displays 07:27:15.0
Console.WriteLine(date1.ToString("hh:mm:ss.F", ci));
// Displays 07:27:15
Console.WriteLine(date1.ToString("hh:mm:ss.ff", ci));
// Displays 07:27:15.01
Console.WriteLine(date1.ToString("hh:mm:ss.FF", ci));
// Displays 07:27:15.01
Console.WriteLine(date1.ToString("hh:mm:ss.fff", ci));
// Displays 07:27:15.018
Console.WriteLine(date1.ToString("hh:mm:ss.FFF", ci));
// Displays 07:27:15.018
```

[Back to table](#)

The "fff" custom format specifier

The "fff" custom format specifier represents the three most significant digits of the seconds fraction; that is, it represents the milliseconds in a date and time value.

The following example includes the "fff" custom format specifier in a custom format string.

C#

```
DateTime date1 = new DateTime(2008, 8, 29, 19, 27, 15, 18);
CultureInfo ci = CultureInfo.InvariantCulture;

Console.WriteLine(date1.ToString("hh:mm:ss.f", ci));
// Displays 07:27:15.0
Console.WriteLine(date1.ToString("hh:mm:ss.F", ci));
// Displays 07:27:15
Console.WriteLine(date1.ToString("hh:mm:ss.ff", ci));
// Displays 07:27:15.01
Console.WriteLine(date1.ToString("hh:mm:ss.FF", ci));
// Displays 07:27:15.01
Console.WriteLine(date1.ToString("hh:mm:ss.fff", ci));
// Displays 07:27:15.018
Console.WriteLine(date1.ToString("hh:mm:ss.FFF", ci));
// Displays 07:27:15.018
```

[Back to table](#)

The "ffff" custom format specifier

The "ffff" custom format specifier represents the four most significant digits of the seconds fraction; that is, it represents the ten thousandths of a second in a date and time value.

Although it's possible to display the ten thousandths of a second component of a time value, that value may not be meaningful. The precision of date and time values depends on the resolution of the system clock. On the Windows NT version 3.5 (and later) and Windows Vista operating systems, the clock's resolution is approximately 10-15 milliseconds.

[Back to table](#)

The "fffff" custom format specifier

The "fffff" custom format specifier represents the five most significant digits of the seconds fraction; that is, it represents the hundred thousandths of a second in a date and time value.

Although it's possible to display the hundred thousandths of a second component of a time value, that value may not be meaningful. The precision of date and time values depends on the resolution of the system clock. On the Windows NT 3.5 (and later) and Windows Vista operating systems, the clock's resolution is approximately 10-15 milliseconds.

[Back to table](#)

The "ffffff" custom format specifier

The "ffffff" custom format specifier represents the six most significant digits of the seconds fraction; that is, it represents the millionths of a second in a date and time value.

Although it's possible to display the millionths of a second component of a time value, that value may not be meaningful. The precision of date and time values depends on the resolution of the system clock. On the Windows NT 3.5 (and later) and Windows Vista operating systems, the clock's resolution is approximately 10-15 milliseconds.

[Back to table](#)

The "fffffff" custom format specifier

The "fffffff" custom format specifier represents the seven most significant digits of the seconds fraction; that is, it represents the ten millionths of a second in a date and time value.

Although it's possible to display the ten millionths of a second component of a time value, that value may not be meaningful. The precision of date and time values depends on the resolution of the system clock. On the Windows NT 3.5 (and later) and Windows Vista operating systems, the clock's resolution is approximately 10-15 milliseconds.

[Back to table](#)

Uppercase seconds "F" fraction specifier

The "F" custom format specifier

The "F" custom format specifier represents the most significant digit of the seconds fraction; that is, it represents the tenths of a second in a date and time value. Nothing is displayed if the digit is zero, and the decimal point that follows the number of seconds is also not displayed.

If the "F" format specifier is used without other format specifiers, it's interpreted as the "F" standard date and time format specifier. For more information about using a single format specifier, see [Using Single Custom Format Specifiers](#) later in this article.

The number of "F" format specifiers used with the [ParseExact](#), [TryParseExact](#), [ParseExact](#), or [TryParseExact](#) method indicates the maximum number of most significant digits of the seconds fraction that can be present to successfully parse the string.

The following example includes the "F" custom format specifier in a custom format string.

C#

```
DateTime date1 = new DateTime(2008, 8, 29, 19, 27, 15, 18);
CultureInfo ci = CultureInfo.InvariantCulture;

Console.WriteLine(date1.ToString("hh:mm:ss.f", ci));
// Displays 07:27:15.0
Console.WriteLine(date1.ToString("hh:mm:ss.F", ci));
// Displays 07:27:15
Console.WriteLine(date1.ToString("hh:mm:ss.fff", ci));
// Displays 07:27:15.01
Console.WriteLine(date1.ToString("hh:mm:ss.FFF", ci));
// Displays 07:27:15.018
Console.WriteLine(date1.ToString("hh:mm:ss.FFF", ci));
// Displays 07:27:15.018
```

[Back to table](#)

The "FF" custom format specifier

The "FF" custom format specifier represents the two most significant digits of the seconds fraction; that is, it represents the hundredths of a second in a date and time value. Trailing zeros aren't displayed. Nothing is displayed if the two significant digits are zero, and in that case the decimal point that follows the number of seconds is also not displayed.

The following example includes the "FF" custom format specifier in a custom format string.

C#

```
DateTime date1 = new DateTime(2008, 8, 29, 19, 27, 15, 18);
CultureInfo ci = CultureInfo.InvariantCulture;

Console.WriteLine(date1.ToString("hh:mm:ss.f", ci));
// Displays 07:27:15.0
Console.WriteLine(date1.ToString("hh:mm:ss.F", ci));
// Displays 07:27:15
Console.WriteLine(date1.ToString("hh:mm:ss.ff", ci));
// Displays 07:27:15.01
Console.WriteLine(date1.ToString("hh:mm:ss.FF", ci));
// Displays 07:27:15.01
Console.WriteLine(date1.ToString("hh:mm:ss.fff", ci));
// Displays 07:27:15.018
Console.WriteLine(date1.ToString("hh:mm:ss.FFF", ci));
// Displays 07:27:15.018
```

[Back to table](#)

The "FFF" custom format specifier

The "FFF" custom format specifier represents the three most significant digits of the seconds fraction; that is, it represents the milliseconds in a date and time value. Trailing zeros aren't displayed. Nothing is displayed if the three significant digits are zero, and in that case the decimal point that follows the number of seconds is also not displayed.

The following example includes the "FFF" custom format specifier in a custom format string.

C#

```
DateTime date1 = new DateTime(2008, 8, 29, 19, 27, 15, 18);
CultureInfo ci = CultureInfo.InvariantCulture;

Console.WriteLine(date1.ToString("hh:mm:ss.f", ci));
// Displays 07:27:15.0
Console.WriteLine(date1.ToString("hh:mm:ss.F", ci));
// Displays 07:27:15
Console.WriteLine(date1.ToString("hh:mm:ss.ff", ci));
// Displays 07:27:15.01
Console.WriteLine(date1.ToString("hh:mm:ss.FF", ci));
// Displays 07:27:15.01
Console.WriteLine(date1.ToString("hh:mm:ss.fff", ci));
// Displays 07:27:15.018
Console.WriteLine(date1.ToString("hh:mm:ss.FFF", ci));
// Displays 07:27:15.018
```

[Back to table](#)

The "FFFF" custom format specifier

The "FFFF" custom format specifier represents the four most significant digits of the seconds fraction; that is, it represents the ten thousandths of a second in a date and time value. Trailing zeros aren't displayed. Nothing is displayed if the four significant digits are zero, and in that case the decimal point that follows the number of seconds is also not displayed.

Although it's possible to display the ten thousandths of a second component of a time value, that value may not be meaningful. The precision of date and time values depends on the resolution of the system clock. On the Windows NT 3.5 (and later) and Windows Vista operating systems, the clock's resolution is approximately 10-15 milliseconds.

[Back to table](#)

The "FFFFFF" custom format specifier

The "FFFFFF" custom format specifier represents the five most significant digits of the seconds fraction; that is, it represents the hundred thousandths of a second in a date and time value. Trailing zeros aren't displayed. Nothing is displayed if the five significant digits are zero, and in that case the decimal point that follows the number of seconds is also not displayed.

Although it's possible to display the hundred thousandths of a second component of a time value, that value may not be meaningful. The precision of date and time values depends on the resolution of the system clock. On the Windows NT 3.5 (and later) and Windows Vista operating systems, the clock's resolution is approximately 10-15 milliseconds.

[Back to table](#)

The "FFFFFFF" custom format specifier

The "FFFFFFF" custom format specifier represents the six most significant digits of the seconds fraction; that is, it represents the millionths of a second in a date and time value. Trailing zeros aren't displayed. Nothing is displayed if the six significant digits are zero, and in that case the decimal point that follows the number of seconds is also not displayed.

Although it's possible to display the millionths of a second component of a time value, that value may not be meaningful. The precision of date and time values depends on the resolution of the system clock. On the Windows NT 3.5 (and later) and Windows Vista operating systems, the clock's resolution is approximately 10-15 milliseconds.

[Back to table](#)

The "FFFFFF" custom format specifier

The "FFFFFF" custom format specifier represents the seven most significant digits of the seconds fraction; that is, it represents the ten millionths of a second in a date and time value. Trailing zeros aren't displayed. Nothing is displayed if the seven significant digits are zero, and in that case the decimal point that follows the number of seconds is also not displayed.

Although it's possible to display the ten millionths of a second component of a time value, that value may not be meaningful. The precision of date and time values depends on the resolution of the system clock. On the Windows NT 3.5 (and later) and Windows Vista operating systems, the clock's resolution is approximately 10-15 milliseconds.

[Back to table](#)

Era "g" format specifier

The "g" or "gg" custom format specifier

The "g" or "gg" custom format specifiers (plus any number of additional "g" specifiers) represents the period or era, such as A.D. The formatting operation ignores this specifier if the date to be formatted doesn't have an associated period or era string.

If the "g" format specifier is used without other custom format specifiers, it's interpreted as the "g" standard date and time format specifier. For more information about using a single format specifier, see [Using Single Custom Format Specifiers](#) later in this article.

The following example includes the "g" custom format specifier in a custom format string.

C#

```
DateTime date1 = new DateTime(70, 08, 04);

Console.WriteLine(date1.ToString("MM/dd/yyyy g",
    CultureInfo.InvariantCulture));
```

```
// Displays 08/04/0070 A.D.  
Console.WriteLine(date1.ToString("MM/dd/yyyy g",  
    CultureInfo.CreateSpecificCulture("fr-FR")));  
// Displays 08/04/0070 ap. J.-C.
```

[Back to table](#)

Lowercase hour "h" format specifier

The "h" custom format specifier

The "h" custom format specifier represents the hour as a number from 1 to 12; that is, the hour is represented by a 12-hour clock that counts the whole hours since midnight or noon. A particular hour after midnight is indistinguishable from the same hour after noon. The hour is not rounded, and a single-digit hour is formatted without a leading zero. For example, given a time of 5:43 in the morning or afternoon, this custom format specifier displays "5".

If the "h" format specifier is used without other custom format specifiers, it's interpreted as a standard date and time format specifier and throws a [FormatException](#). For more information about using a single format specifier, see [Using Single Custom Format Specifiers](#) later in this article.

The following example includes the "h" custom format specifier in a custom format string.

C#

```
DateTime date1;  
date1 = new DateTime(2008, 1, 1, 18, 9, 1);  
Console.WriteLine(date1.ToString("h:m:s.F t",  
    CultureInfo.InvariantCulture));  
// Displays 6:9:1 P  
Console.WriteLine(date1.ToString("h:m:s.F t",  
    CultureInfo.CreateSpecificCulture("el-GR")));  
// Displays 6:9:1 μ  
date1 = new DateTime(2008, 1, 1, 18, 9, 1, 500);  
Console.WriteLine(date1.ToString("h:m:s.F t",  
    CultureInfo.InvariantCulture));  
// Displays 6:9:1.5 P  
Console.WriteLine(date1.ToString("h:m:s.F t",  
    CultureInfo.CreateSpecificCulture("el-GR")));  
// Displays 6:9:1.5 μ
```

[Back to table](#)

The "hh" custom format specifier

The "hh" custom format specifier (plus any number of additional "h" specifiers) represents the hour as a number from 01 to 12; that is, the hour is represented by a 12-hour clock that counts the whole hours since midnight or noon. A particular hour after midnight is indistinguishable from the same hour after noon. The hour is not rounded, and a single-digit hour is formatted with a leading zero. For example, given a time of 5:43 in the morning or afternoon, this format specifier displays "05".

The following example includes the "hh" custom format specifier in a custom format string.

C#

```
DateTime date1;
date1 = new DateTime(2008, 1, 1, 18, 9, 1);
Console.WriteLine(date1.ToString("hh:mm:ss tt",
                               CultureInfo.InvariantCulture));
// Displays 06:09:01 PM
Console.WriteLine(date1.ToString("hh:mm:ss tt",
                               CultureInfo.CreateSpecificCulture("hu-HU")));
// Displays 06:09:01 du.
date1 = new DateTime(2008, 1, 1, 18, 9, 1, 500);
Console.WriteLine(date1.ToString("hh:mm:ss.ff tt",
                               CultureInfo.InvariantCulture));
// Displays 06:09:01.50 PM
Console.WriteLine(date1.ToString("hh:mm:ss.ff tt",
                               CultureInfo.CreateSpecificCulture("hu-HU")));
// Displays 06:09:01.50 du.
```

[Back to table](#)

Uppercase hour "H" format specifier

The "H" custom format specifier

The "H" custom format specifier represents the hour as a number from 0 to 23; that is, the hour is represented by a zero-based 24-hour clock that counts the hours since midnight. A single-digit hour is formatted without a leading zero.

If the "H" format specifier is used without other custom format specifiers, it's interpreted as a standard date and time format specifier and throws a [FormatException](#). For more information about using a single format specifier, see [Using Single Custom Format Specifiers](#) later in this article.

The following example includes the "H" custom format specifier in a custom format string.

```
C#
```

```
DateTime date1 = new DateTime(2008, 1, 1, 6, 9, 1);
Console.WriteLine(date1.ToString("H:mm:ss",
    CultureInfo.InvariantCulture));
// Displays 6:09:01
```

[Back to table](#)

The "HH" custom format specifier

The "HH" custom format specifier (plus any number of additional "H" specifiers) represents the hour as a number from 00 to 23; that is, the hour is represented by a zero-based 24-hour clock that counts the hours since midnight. A single-digit hour is formatted with a leading zero.

The following example includes the "HH" custom format specifier in a custom format string.

```
C#
```

```
DateTime date1 = new DateTime(2008, 1, 1, 6, 9, 1);
Console.WriteLine(date1.ToString("HH:mm:ss",
    CultureInfo.InvariantCulture));
// Displays 06:09:01
```

[Back to table](#)

Time zone "K" format specifier

The "K" custom format specifier

The "K" custom format specifier represents the time zone information of a date and time value. When this format specifier is used with [DateTime](#) values, the result string is defined by the value of the [DateTime.Kind](#) property:

- For the local time zone (a [DateTime.Kind](#) property value of [DateTimeKind.Local](#)), this specifier produces a result string containing the local offset from Coordinated Universal Time (UTC); for example, "-07:00".

- For a UTC time (a `DateTime.Kind` property value of `DateTimeKind.Utc`), the result string includes a "Z" character to represent a UTC date.
- For a time from an unspecified time zone (a time whose `DateTime.Kind` property equals `DateTimeKind.Unspecified`), the result is equivalent to `String.Empty`.

For `DateTimeOffset` values, the "K" format specifier is equivalent to the "zzz" format specifier, and produces a result string containing the `DateTimeOffset` value's offset from UTC.

If the "K" format specifier is used without other custom format specifiers, it's interpreted as a standard date and time format specifier and throws a `FormatException`. For more information about using a single format specifier, see [Using Single Custom Format Specifiers](#) later in this article.

The following example displays the string that results from using the "K" custom format specifier with various `DateTime` and `DateTimeOffset` values on a system in the U.S. Pacific Time zone.

C#

```
Console.WriteLine(DateTime.Now.ToString("%K"));
// Displays -07:00
Console.WriteLine(DateTime.UtcNow.ToString("%K"));
// Displays Z
Console.WriteLine("'{0}'",
    DateTime.SpecifyKind(DateTime.Now,
        DateTimeKind.Unspecified).ToString("%K"));
// Displays ''
Console.WriteLine(DateTimeOffset.Now.ToString("%K"));
// Displays -07:00
Console.WriteLine(DateTimeOffset.UtcNow.ToString("%K"));
// Displays +00:00
Console.WriteLine(new DateTimeOffset(2008, 5, 1, 6, 30, 0,
    new TimeSpan(5, 0, 0)).ToString("%K"));
// Displays +05:00
```

[Back to table](#)

Minute "m" format specifier

The "m" custom format specifier

The "m" custom format specifier represents the minute as a number from 0 to 59. The minute represents whole minutes that have passed since the last hour. A single-digit

minute is formatted without a leading zero.

If the "m" format specifier is used without other custom format specifiers, it's interpreted as the "m" standard date and time format specifier. For more information about using a single format specifier, see [Using Single Custom Format Specifiers](#) later in this article.

The following example includes the "m" custom format specifier in a custom format string.

C#

```
DateTime date1;
date1 = new DateTime(2008, 1, 1, 18, 9, 1);
Console.WriteLine(date1.ToString("h:m:s.F t",
                               CultureInfo.InvariantCulture));
// Displays 6:9:1 P
Console.WriteLine(date1.ToString("h:m:s.F t",
                               CultureInfo.CreateSpecificCulture("el-GR")));
// Displays 6:9:1 μ
date1 = new DateTime(2008, 1, 1, 18, 9, 1, 500);
Console.WriteLine(date1.ToString("h:m:s.F t",
                               CultureInfo.InvariantCulture));
// Displays 6:9:1.5 P
Console.WriteLine(date1.ToString("h:m:s.F t",
                               CultureInfo.CreateSpecificCulture("el-GR")));
// Displays 6:9:1.5 μ
```

[Back to table](#)

The "mm" custom format specifier

The "mm" custom format specifier (plus any number of additional "m" specifiers) represents the minute as a number from 00 to 59. The minute represents whole minutes that have passed since the last hour. A single-digit minute is formatted with a leading zero.

The following example includes the "mm" custom format specifier in a custom format string.

C#

```
DateTime date1;
date1 = new DateTime(2008, 1, 1, 18, 9, 1);
Console.WriteLine(date1.ToString("hh:mm:ss tt",
                               CultureInfo.InvariantCulture));
// Displays 06:09:01 PM
Console.WriteLine(date1.ToString("hh:mm:ss tt",
                               CultureInfo.CreateSpecificCulture("hu-HU")));
```

```
// Displays 06:09:01 du.
date1 = new DateTime(2008, 1, 1, 18, 9, 1, 500);
Console.WriteLine(date1.ToString("hh:mm:ss.ff tt",
    CultureInfo.InvariantCulture));
// Displays 06:09:01.50 PM
Console.WriteLine(date1.ToString("hh:mm:ss.ff tt",
    CultureInfo.CreateSpecificCulture("hu-HU")));
// Displays 06:09:01.50 du.
```

[Back to table](#)

Month "M" format specifier

The "M" custom format specifier

The "M" custom format specifier represents the month as a number from 1 to 12 (or from 1 to 13 for calendars that have 13 months). A single-digit month is formatted without a leading zero.

If the "M" format specifier is used without other custom format specifiers, it's interpreted as the "M" standard date and time format specifier. For more information about using a single format specifier, see [Using Single Custom Format Specifiers](#) later in this article.

The following example includes the "M" custom format specifier in a custom format string.

C#

```
DateTime date1 = new DateTime(2008, 8, 18);
Console.WriteLine(date1.ToString("(M) MMM, MMMMM",
    CultureInfo.CreateSpecificCulture("en-US")));
// Displays (8) Aug, August
Console.WriteLine(date1.ToString("(M) MMM, MMMMM",
    CultureInfo.CreateSpecificCulture("nl-NL")));
// Displays (8) aug, augustus
Console.WriteLine(date1.ToString("(M) MMM, MMMMM",
    CultureInfo.CreateSpecificCulture("lv-LV")));
// Displays (8) Aug, augusts
```

[Back to table](#)

The "MM" custom format specifier

The "MM" custom format specifier represents the month as a number from 01 to 12 (or from 1 to 13 for calendars that have 13 months). A single-digit month is formatted with a leading zero.

The following example includes the "MM" custom format specifier in a custom format string.

```
C#
```

```
DateTime date1 = new DateTime(2008, 1, 2, 6, 30, 15);

Console.WriteLine(date1.ToString("dd, MM",
    CultureInfo.InvariantCulture));
// 02, 01
```

[Back to table](#)

The "MMM" custom format specifier

The "MMM" custom format specifier represents the abbreviated name of the month.

The localized abbreviated name of the month is retrieved from the [DateTimeFormatInfo.AbbreviatedMonthNames](#) property of the current or specified culture.

The following example includes the "MMM" custom format specifier in a custom format string.

```
C#
```

```
DateTime date1 = new DateTime(2008, 8, 29, 19, 27, 15);

Console.WriteLine(date1.ToString("ddd d MMM",
    CultureInfo.CreateSpecificCulture("en-US")));
// Displays Fri 29 Aug
Console.WriteLine(date1.ToString("ddd d MMM",
    CultureInfo.CreateSpecificCulture("fr-FR")));
// Displays ven. 29 août
```

[Back to table](#)

The "MMMM" custom format specifier

The "MMMM" custom format specifier represents the full name of the month. The localized name of the month is retrieved from the [DateTimeFormatInfo.MonthNames](#) property of the current or specified culture.

The following example includes the "MMMM" custom format specifier in a custom format string.

C#

```
DateTime date1 = new DateTime(2008, 8, 29, 19, 27, 15);

Console.WriteLine(date1.ToString("ddd dd MMMM",
                               CultureInfo.CreateSpecificCulture("en-US")));
// Displays Friday 29 August
Console.WriteLine(date1.ToString("ddd dd MMMM",
                               CultureInfo.CreateSpecificCulture("it-IT")));
// Displays venerdì 29 agosto
```

[Back to table](#)

Seconds "s" format specifier

The "s" custom format specifier

The "s" custom format specifier represents the seconds as a number from 0 to 59. The result represents whole seconds that have passed since the last minute. A single-digit second is formatted without a leading zero.

If the "s" format specifier is used without other custom format specifiers, it's interpreted as the "s" standard date and time format specifier. For more information about using a single format specifier, see [Using Single Custom Format Specifiers](#) later in this article.

The following example includes the "s" custom format specifier in a custom format string.

C#

```
DateTime date1;
date1 = new DateTime(2008, 1, 1, 18, 9, 1);
Console.WriteLine(date1.ToString("h:m:s.F t",
                               CultureInfo.InvariantCulture));
// Displays 6:9:1 P
Console.WriteLine(date1.ToString("h:m:s.F t",
                               CultureInfo.CreateSpecificCulture("el-GR")));
// Displays 6:9:1 μ
date1 = new DateTime(2008, 1, 1, 18, 9, 1, 500);
Console.WriteLine(date1.ToString("h:m:s.F t",
                               CultureInfo.InvariantCulture));
// Displays 6:9:1.5 P
Console.WriteLine(date1.ToString("h:m:s.F t",
```

```
CultureInfo.CreateSpecificCulture("el-GR")));  
// Displays 6:9:1.5 μ
```

[Back to table](#)

The "ss" custom format specifier

The "ss" custom format specifier (plus any number of additional "s" specifiers) represents the seconds as a number from 00 to 59. The result represents whole seconds that have passed since the last minute. A single-digit second is formatted with a leading zero.

The following example includes the "ss" custom format specifier in a custom format string.

C#

```
DateTime date1;  
date1 = new DateTime(2008, 1, 1, 18, 9, 1);  
Console.WriteLine(date1.ToString("hh:mm:ss tt",  
                                CultureInfo.InvariantCulture));  
// Displays 06:09:01 PM  
Console.WriteLine(date1.ToString("hh:mm:ss tt",  
                                CultureInfo.CreateSpecificCulture("hu-HU")));  
// Displays 06:09:01 du.  
date1 = new DateTime(2008, 1, 1, 18, 9, 1, 500);  
Console.WriteLine(date1.ToString("hh:mm:ss.fff tt",  
                                CultureInfo.InvariantCulture));  
// Displays 06:09:01.50 PM  
Console.WriteLine(date1.ToString("hh:mm:ss.fff tt",  
                                CultureInfo.CreateSpecificCulture("hu-HU")));  
// Displays 06:09:01.50 du.
```

[Back to table](#)

Meridiem "t" format specifier

The "t" custom format specifier

The "t" custom format specifier represents the first character of the AM/PM designator. The appropriate localized designator is retrieved from the [DateTimeFormatInfo.AMDesignator](#) or [DateTimeFormatInfo.PMDesignator](#) property of the current or specific culture. The AM designator is used for all times from 0:00:00

(midnight) to 11:59:59.999. The PM designator is used for all times from 12:00:00 (noon) to 23:59:59.999.

If the "t" format specifier is used without other custom format specifiers, it's interpreted as the "t" standard date and time format specifier. For more information about using a single format specifier, see [Using Single Custom Format Specifiers](#) later in this article.

The following example includes the "t" custom format specifier in a custom format string.

C#

```
DateTime date1;
date1 = new DateTime(2008, 1, 1, 18, 9, 1);
Console.WriteLine(date1.ToString("h:m:s.F t",
                               CultureInfo.InvariantCulture));
// Displays 6:9:1 P
Console.WriteLine(date1.ToString("h:m:s.F t",
                               CultureInfo.CreateSpecificCulture("el-GR")));
// Displays 6:9:1 μ
date1 = new DateTime(2008, 1, 1, 18, 9, 1, 500);
Console.WriteLine(date1.ToString("h:m:s.F t",
                               CultureInfo.InvariantCulture));
// Displays 6:9:1.5 P
Console.WriteLine(date1.ToString("h:m:s.F t",
                               CultureInfo.CreateSpecificCulture("el-GR")));
// Displays 6:9:1.5 μ
```

[Back to table](#)

The "tt" custom format specifier

The "tt" custom format specifier (plus any number of additional "t" specifiers) represents the entire AM/PM designator. The appropriate localized designator is retrieved from the [DateTimeFormatInfo.AMDesignator](#) or [DateTimeFormatInfo.PMDesignator](#) property of the current or specific culture. The AM designator is used for all times from 0:00:00 (midnight) to 11:59:59.999. The PM designator is used for all times from 12:00:00 (noon) to 23:59:59.999.

Make sure to use the "tt" specifier for languages for which it's necessary to maintain the distinction between AM and PM. An example is Japanese, for which the AM and PM designators differ in the second character instead of the first character.

The following example includes the "tt" custom format specifier in a custom format string.

C#

```
DateTime date1;
date1 = new DateTime(2008, 1, 1, 18, 9, 1);
Console.WriteLine(date1.ToString("hh:mm:ss tt",
                               CultureInfo.InvariantCulture));
// Displays 06:09:01 PM
Console.WriteLine(date1.ToString("hh:mm:ss tt",
                               CultureInfo.CreateSpecificCulture("hu-HU")));
// Displays 06:09:01 du.
date1 = new DateTime(2008, 1, 1, 18, 9, 1, 500);
Console.WriteLine(date1.ToString("hh:mm:ss.ff tt",
                               CultureInfo.InvariantCulture));
// Displays 06:09:01.50 PM
Console.WriteLine(date1.ToString("hh:mm:ss.ff tt",
                               CultureInfo.CreateSpecificCulture("hu-HU")));
// Displays 06:09:01.50 du.
```

[Back to table](#)

Year "y" format specifier

The "y" custom format specifier

The "y" custom format specifier represents the year as a one-digit or two-digit number. If the year has more than two digits, only the two low-order digits appear in the result. If the first digit of a two-digit year begins with a zero (for example, 2008), the number is formatted without a leading zero.

If the "y" format specifier is used without other custom format specifiers, it's interpreted as the "y" standard date and time format specifier. For more information about using a single format specifier, see [Using Single Custom Format Specifiers](#) later in this article.

The following example includes the "y" custom format specifier in a custom format string.

C#

```
DateTime date1 = new DateTime(1, 12, 1);
DateTime date2 = new DateTime(2010, 1, 1);
Console.WriteLine(date1.ToString("%y"));
// Displays 1
Console.WriteLine(date1.ToString("yy"));
// Displays 01
Console.WriteLine(date1.ToString("yyy"));
// Displays 001
Console.WriteLine(date1.ToString("yyyy"));
```

```

// Displays 0001
Console.WriteLine(date1.ToString("yyyy"));
// Displays 0001
Console.WriteLine(date2.ToString("%y"));
// Displays 10
Console.WriteLine(date2.ToString("yy"));
// Displays 10
Console.WriteLine(date2.ToString("yyy"));
// Displays 2010
Console.WriteLine(date2.ToString("yyyy"));
// Displays 2010
Console.WriteLine(date2.ToString("yyyyy"));
// Displays 02010

```

[Back to table](#)

The "yy" custom format specifier

The "yy" custom format specifier represents the year as a two-digit number. If the year has more than two digits, only the two low-order digits appear in the result. If the two-digit year has fewer than two significant digits, the number is padded with leading zeros to produce two digits.

In a parsing operation, a two-digit year that is parsed using the "yy" custom format specifier is interpreted based on the [Calendar.TwoDigitYearMax](#) property of the format provider's current calendar. The following example parses the string representation of a date that has a two-digit year by using the default Gregorian calendar of the en-US culture, which, in this case, is the current culture. It then changes the current culture's [CultureInfo](#) object to use a [GregorianCalendar](#) object whose [TwoDigitYearMax](#) property has been modified.

C#

```

using System;
using System.Globalization;
using System.Threading;

public class Example7
{
    public static void Main()
    {
        string fmt = "dd-MMM-yy";
        string value = "24-Jan-49";

        Calendar cal =
(Calendar)CultureInfo.CurrentCulture.Calendar.Clone();
        Console.WriteLine("Two Digit Year Range: {0} - {1}",
                           cal.TwoDigitYearMax - 99, cal.TwoDigitYearMax);
    }
}

```

```

        Console.WriteLine("{0:d}", DateTime.ParseExact(value, fmt, null));
        Console.WriteLine();

        cal.TwoDigitYearMax = 2099;
        CultureInfo culture =
(CultureInfo)CultureInfo.CurrentCulture.Clone();
        culture.DateTimeFormat.Calendar = cal;
        Thread.CurrentThread.CurrentCulture = culture;

        Console.WriteLine("Two Digit Year Range: {0} - {1}",
                           cal.TwoDigitYearMax - 99, cal.TwoDigitYearMax);
        Console.WriteLine("{0:d}", DateTime.ParseExact(value, fmt, null));
    }
}

// The example displays the following output:
//      Two Digit Year Range: 1930 - 2029
//      1/24/1949
//
//      Two Digit Year Range: 2000 - 2099
//      1/24/2049

```

The following example includes the "yy" custom format specifier in a custom format string.

C#

```

DateTime date1 = new DateTime(1, 12, 1);
DateTime date2 = new DateTime(2010, 1, 1);
Console.WriteLine(date1.ToString("%y"));
// Displays 1
Console.WriteLine(date1.ToString("yy"));
// Displays 01
Console.WriteLine(date1.ToString("yyy"));
// Displays 001
Console.WriteLine(date1.ToString("yyyy"));
// Displays 0001
Console.WriteLine(date1.ToString("yyyyy"));
// Displays 00001
Console.WriteLine(date2.ToString("%y"));
// Displays 10
Console.WriteLine(date2.ToString("yy"));
// Displays 10
Console.WriteLine(date2.ToString("yyy"));
// Displays 2010
Console.WriteLine(date2.ToString("yyyy"));
// Displays 2010
Console.WriteLine(date2.ToString("yyyyy"));
// Displays 02010

```

[Back to table](#)

The "yyy" custom format specifier

The "yyy" custom format specifier represents the year with a minimum of three digits. If the year has more than three significant digits, they are included in the result string. If the year has fewer than three digits, the number is padded with leading zeros to produce three digits.

ⓘ Note

For the Thai Buddhist calendar, which can have five-digit years, this format specifier displays all significant digits.

The following example includes the "yyy" custom format specifier in a custom format string.

C#

```
DateTime date1 = new DateTime(1, 12, 1);
DateTime date2 = new DateTime(2010, 1, 1);
Console.WriteLine(date1.ToString("%y"));
// Displays 1
Console.WriteLine(date1.ToString("yy"));
// Displays 01
Console.WriteLine(date1.ToString("yyy"));
// Displays 001
Console.WriteLine(date1.ToString("yyyy"));
// Displays 0001
Console.WriteLine(date1.ToString("yyyyy"));
// Displays 00001
Console.WriteLine(date2.ToString("%y"));
// Displays 10
Console.WriteLine(date2.ToString("yy"));
// Displays 10
Console.WriteLine(date2.ToString("yyy"));
// Displays 2010
Console.WriteLine(date2.ToString("yyyy"));
// Displays 2010
Console.WriteLine(date2.ToString("yyyyy"));
// Displays 02010
```

[Back to table](#)

The "yyyy" custom format specifier

The "yyyy" custom format specifier represents the year with a minimum of four digits. If the year has more than four significant digits, they are included in the result string. If the

year has fewer than four digits, the number is padded with leading zeros to produce four digits.

ⓘ Note

For the Thai Buddhist calendar, which can have five-digit years, this format specifier displays a minimum of four digits.

The following example includes the "yyyy" custom format specifier in a custom format string.

C#

```
DateTime date1 = new DateTime(1, 12, 1);
DateTime date2 = new DateTime(2010, 1, 1);
Console.WriteLine(date1.ToString("%y"));
// Displays 1
Console.WriteLine(date1.ToString("yy"));
// Displays 01
Console.WriteLine(date1.ToString("yyy"));
// Displays 001
Console.WriteLine(date1.ToString("yyyy"));
// Displays 0001
Console.WriteLine(date1.ToString("yyyyy"));
// Displays 00001
Console.WriteLine(date2.ToString("%y"));
// Displays 10
Console.WriteLine(date2.ToString("yy"));
// Displays 10
Console.WriteLine(date2.ToString("yyy"));
// Displays 2010
Console.WriteLine(date2.ToString("yyyy"));
// Displays 2010
Console.WriteLine(date2.ToString("yyyyy"));
// Displays 02010
```

[Back to table](#)

The "yyyyy" custom format specifier

The "yyyyy" custom format specifier (plus any number of additional "y" specifiers) represents the year with a minimum of five digits. If the year has more than five significant digits, they are included in the result string. If the year has fewer than five digits, the number is padded with leading zeros to produce five digits.

If there are additional "y" specifiers, the number is padded with as many leading zeros as necessary to produce the number of "y" specifiers.

The following example includes the "yyyyy" custom format specifier in a custom format string.

C#

```
DateTime date1 = new DateTime(1, 12, 1);
DateTime date2 = new DateTime(2010, 1, 1);
Console.WriteLine(date1.ToString("%y"));
// Displays 1
Console.WriteLine(date1.ToString("yy"));
// Displays 01
Console.WriteLine(date1.ToString("yyy"));
// Displays 001
Console.WriteLine(date1.ToString("yyyy"));
// Displays 0001
Console.WriteLine(date1.ToString("yyyyy"));
// Displays 00001
Console.WriteLine(date2.ToString("%y"));
// Displays 10
Console.WriteLine(date2.ToString("yy"));
// Displays 10
Console.WriteLine(date2.ToString("yyy"));
// Displays 2010
Console.WriteLine(date2.ToString("yyyy"));
// Displays 2010
Console.WriteLine(date2.ToString("yyyyy"));
// Displays 02010
```

[Back to table](#)

Offset "z" format specifier

The "z" custom format specifier

With [DateTime](#) values, the "z" custom format specifier represents the signed offset of the specified time zone from Coordinated Universal Time (UTC), measured in hours. The offset is always displayed with a leading sign. A plus sign (+) indicates hours ahead of UTC, and a minus sign (-) indicates hours behind UTC. A single-digit offset is formatted *without* a leading zero.

The following table shows how the offset value changes depending on [DateTimeKind](#).

| DateTimeKind | Offset value |
|---------------------|--|
| Local | The signed offset of the local operating system's time zone from UTC. |
| Unspecified | The signed offset of the local operating system's time zone from UTC. |
| Utc | +0 on .NET Core and .NET 5+. On .NET Framework, the signed offset of the local operating system's time zone from UTC. |

With [DateTimeOffset](#) values, this format specifier represents the [DateTimeOffset](#) value's offset from UTC in hours.

If the "z" format specifier is used without other custom format specifiers, it's interpreted as a standard date and time format specifier and throws a [FormatException](#). For more information about using a single format specifier, see [Using Single Custom Format Specifiers](#) later in this article.

The following example includes the "z" custom format specifier in a custom format string.

C#

```
DateTime date1 = DateTime.UtcNow;
Console.WriteLine(String.Format("{0:%z}, {0:zz}, {0:zzz}",
    date1));
// Displays -7, -07, -07:00 on .NET Framework
// Displays +0, +00, +00:00 on .NET Core and .NET 5+

DateTimeOffset date2 = new DateTimeOffset(2008, 8, 1, 0, 0, 0,
    new TimeSpan(6, 0, 0));
Console.WriteLine(String.Format("{0:%z}, {0:zz}, {0:zzz}",
    date2));
// Displays +6, +06, +06:00
```

[Back to table](#)

The "zz" custom format specifier

With [DateTime](#) values, the "zz" custom format specifier represents the signed offset of the specified time zone from UTC, measured in hours. The offset is always displayed with a leading sign. A plus sign (+) indicates hours ahead of UTC, and a minus sign (-) indicates hours behind UTC. A single-digit offset is formatted *with* a leading zero.

The following table shows how the offset value changes depending on [DateTimeKind](#).

| DateTimeKind | Offset value |
|---------------------|---|
| Local | The signed offset of the local operating system's time zone from UTC. |
| Unspecified | The signed offset of the local operating system's time zone from UTC. |
| Utc | +00 on .NET Core and .NET 5+. On .NET Framework, the signed offset of the local operating system's time zone from UTC. |

With [DateTimeOffset](#) values, this format specifier represents the [DateTimeOffset](#) value's offset from UTC in hours.

The following example includes the "zz" custom format specifier in a custom format string.

C#

```
DateTime date1 = DateTime.UtcNow;
Console.WriteLine(String.Format("{0:%z}, {0:zz}, {0:zzz}",
                               date1));
// Displays -7, -07, -07:00 on .NET Framework
// Displays +0, +00, +00:00 on .NET Core and .NET 5+

DateTimeOffset date2 = new DateTimeOffset(2008, 8, 1, 0, 0, 0,
                                         new TimeSpan(6, 0, 0));
Console.WriteLine(String.Format("{0:%z}, {0:zz}, {0:zzz}",
                               date2));
// Displays +6, +06, +06:00
```

[Back to table](#)

The "zzz" custom format specifier

With [DateTime](#) values, the "zzz" custom format specifier represents the signed offset of the specified time zone from UTC, measured in hours and minutes. The offset is always displayed with a leading sign. A plus sign (+) indicates hours ahead of UTC, and a minus sign (-) indicates hours behind UTC. A single-digit offset is formatted with a leading zero.

The following table shows how the offset value changes depending on [DateTimeKind](#).

| DateTimeKind | Offset value |
|---------------------|--|
| Local | The signed offset of the local operating system's time zone from UTC. |
| Unspecified | The signed offset of the local operating system's time zone from UTC. |
| Utc | +00:00 on .NET Core and .NET 5+. On .NET Framework, the signed offset of the local operating system's time zone from UTC. |

With [DateTimeOffset](#) values, this format specifier represents the [DateTimeOffset](#) value's offset from UTC in hours and minutes.

The following example includes the "zzz" custom format specifier in a custom format string.

C#

```
DateTime date1 = DateTime.UtcNow;
Console.WriteLine(String.Format("{0:%z}, {0:zz}, {0:zzz}",
                               date1));
// Displays -7, -07, -07:00 on .NET Framework
// Displays +0, +00, +00:00 on .NET Core and .NET 5+

DateTimeOffset date2 = new DateTimeOffset(2008, 8, 1, 0, 0, 0,
                                         new TimeSpan(6, 0, 0));
Console.WriteLine(String.Format("{0:%z}, {0:zz}, {0:zzz}",
                               date2));
// Displays +6, +06, +06:00
```

[Back to table](#)

Date and time separator specifiers

The ":" custom format specifier

The ":" custom format specifier represents the time separator, which is used to differentiate hours, minutes, and seconds. The appropriate localized time separator is retrieved from the [DateTimeFormatInfo.TimeSeparator](#) property of the current or specified culture.

 **Note**

To change the time separator for a particular date and time string, specify the separator character within a literal string delimiter. For example, the custom format string `hh'_'dd'_'ss` produces a result string in which "_" (an underscore) is always used as the time separator. To change the time separator for all dates for a culture, either change the value of the `DateTimeFormatInfo.TimeSeparator` property of the current culture, or instantiate a `DateTimeFormatInfo` object, assign the character to its `TimeSeparator` property, and call an overload of the formatting method that includes an `IFormatProvider` parameter.

If the ":" format specifier is used without other custom format specifiers, it's interpreted as a standard date and time format specifier and throws a `FormatException`. For more information about using a single format specifier, see [Using Single Custom Format Specifiers](#) later in this article.

[Back to table](#)

The "/" custom format specifier

The "/" custom format specifier represents the date separator, which is used to differentiate years, months, and days. The appropriate localized date separator is retrieved from the `DateTimeFormatInfo.DateSeparator` property of the current or specified culture.

ⓘ Note

To change the date separator for a particular date and time string, specify the separator character within a literal string delimiter. For example, the custom format string `mm'/'dd'/'yyyy` produces a result string in which "/" is always used as the date separator. To change the date separator for all dates for a culture, either change the value of the `DateTimeFormatInfo.DateSeparator` property of the current culture, or instantiate a `DateTimeFormatInfo` object, assign the character to its `DateSeparator` property, and call an overload of the formatting method that includes an `IFormatProvider` parameter.

If the "/" format specifier is used without other custom format specifiers, it's interpreted as a standard date and time format specifier and throws a `FormatException`. For more information about using a single format specifier, see [Using Single Custom Format Specifiers](#) later in this article.

[Back to table](#)

Character literals

The following characters in a custom date and time format string are reserved and are always interpreted as formatting characters or, in the case of `"`, `'`, `/`, and `\`, as special characters.

- `F`
- `H`
- `K`
- `M`
- `d`
- `f`
- `g`
- `h`
- `m`
- `s`
- `t`
- `y`
- `z`
- `%`
- `:`
- `/`
- `"`
- `'`
- `\`

All other characters are always interpreted as character literals and, in a formatting operation, are included in the result string unchanged. In a parsing operation, they must match the characters in the input string exactly; the comparison is case-sensitive.

The following example includes the literal characters "PST" (for Pacific Standard Time) and "PDT" (for Pacific Daylight Time) to represent the local time zone in a format string. Note that the string is included in the result string, and that a string that includes the local time zone string also parses successfully.

C#

```
using System;
using System.Globalization;

public class Example5
{
```

```

public static void Main()
{
    String[] formats = { "dd MMM yyyy hh:mm tt PST",
                        "dd MMM yyyy hh:mm tt PDT" };
    var dat = new DateTime(2016, 8, 18, 16, 50, 0);
    // Display the result string.
    Console.WriteLine(dat.ToString(formats[1]));

    // Parse a string.
    String value = "25 Dec 2016 12:00 pm PST";
    DateTime newDate;
    if (DateTime.TryParseExact(value, formats, null,
                               DateTimeStyles.None, out newDate))
        Console.WriteLine(newDate);
    else
        Console.WriteLine("Unable to parse '{0}'", value);
}
// The example displays the following output:
//      18 Aug 2016 04:50 PM PDT
//      12/25/2016 12:00:00 PM

```

There are two ways to indicate that characters are to be interpreted as literal characters and not as reserve characters, so that they can be included in a result string or successfully parsed in an input string:

- By escaping each reserved character. For more information, see [Using the Escape Character](#).

The following example includes the literal characters "pst" (for Pacific Standard time) to represent the local time zone in a format string. Because both "s" and "t" are custom format strings, both characters must be escaped to be interpreted as character literals.

C#

```

using System;
using System.Globalization;

public class Example3
{
    public static void Main()
    {
        String format = "dd MMM yyyy hh:mm tt p\\s\\t";
        var dat = new DateTime(2016, 8, 18, 16, 50, 0);
        // Display the result string.
        Console.WriteLine(dat.ToString(format));

        // Parse a string.
        String value = "25 Dec 2016 12:00 pm pst";
        DateTime newDate;
        if (DateTime.TryParseExact(value, format, null,

```

```

                DateTimeStyles.None, out newDate))
        Console.WriteLine(newDate);
    else
        Console.WriteLine("Unable to parse '{0}'", value);
}
// The example displays the following output:
//      18 Aug 2016 04:50 PM pst
//      12/25/2016 12:00:00 PM

```

- By enclosing the entire literal string in quotation marks or apostrophes. The following example is like the previous one, except that "pst" is enclosed in quotation marks to indicate that the entire delimited string should be interpreted as character literals.

C#

```

using System;
using System.Globalization;

public class Example6
{
    public static void Main()
    {
        String format = "dd MMM yyyy hh:mm tt \"pst\"";
        var dat = new DateTime(2016, 8, 18, 16, 50, 0);
        // Display the result string.
        Console.WriteLine(dat.ToString(format));

        // Parse a string.
        String value = "25 Dec 2016 12:00 pm pst";
        DateTime newDate;
        if (DateTime.TryParseExact(value, format, null,
            DateTimeStyles.None, out newDate))
            Console.WriteLine(newDate);
        else
            Console.WriteLine("Unable to parse '{0}'", value);
    }
}
// The example displays the following output:
//      18 Aug 2016 04:50 PM pst
//      12/25/2016 12:00:00 PM

```

Notes

Using single custom format specifiers

A custom date and time format string consists of two or more characters. Date and time formatting methods interpret any single-character string as a standard date and time format string. If they don't recognize the character as a valid format specifier, they throw a [FormatException](#). For example, a format string that consists only of the specifier "h" is interpreted as a standard date and time format string. However, in this particular case, an exception is thrown because there is no "h" standard date and time format specifier.

To use any of the custom date and time format specifiers as the only specifier in a format string (that is, to use the "d", "f", "F", "g", "h", "H", "K", "m", "M", "s", "t", "y", "z", ":", or "/" custom format specifier by itself), include a space before or after the specifier, or include a percent ("%") format specifier before the single custom date and time specifier.

For example, "%h" is interpreted as a custom date and time format string that displays the hour represented by the current date and time value. You can also use the " h" or "h" format string, although this includes a space in the result string along with the hour. The following example illustrates these three format strings.

C#

```
DateTime dat1 = new DateTime(2009, 6, 15, 13, 45, 0);

Console.WriteLine("{0:%h}", dat1);
Console.WriteLine("{0: h}", dat1);
Console.WriteLine("{0:h }", dat1);
// The example displays the following output:
//      '1'
//      ' 1'
//      '1 '
```

Using the Escape character

The "d", "f", "F", "g", "h", "H", "K", "m", "M", "s", "t", "y", "z", ":", or "/" characters in a format string are interpreted as custom format specifiers rather than as literal characters. To prevent a character from being interpreted as a format specifier, you can precede it with a backslash (\), which is the escape character. The escape character signifies that the following character is a character literal that should be included in the result string unchanged.

To include a backslash in a result string, you must escape it with another backslash (\\\).

 **Note**

Some compilers, such as the C++ and C# compilers, may also interpret a single backslash character as an escape character. To ensure that a string is interpreted correctly when formatting, you can use the verbatim string literal character (the @ character) before the string in C#, or add another backslash character before each backslash in C# and C++. The following C# example illustrates both approaches.

The following example uses the escape character to prevent the formatting operation from interpreting the "h" and "m" characters as format specifiers.

C#

```
DateTime date = new DateTime(2009, 06, 15, 13, 45, 30, 90);
string fmt1 = "h \\h m \\m";
string fmt2 = @"h \h m \m";

Console.WriteLine("{0} ({1}) -> {2}", date, fmt1, date.ToString(fmt1));
Console.WriteLine("{0} ({1}) -> {2}", date, fmt2, date.ToString(fmt2));
// The example displays the following output:
//      6/15/2009 1:45:30 PM (h \h m \m) -> 1 h 45 m
//      6/15/2009 1:45:30 PM (h \h m \m) -> 1 h 45 m
```

Control Panel settings

The **Regional and Language Options** settings in Control Panel influence the result string produced by a formatting operation that includes many of the custom date and time format specifiers. These settings are used to initialize the [DateTimeFormatInfo](#) object associated with the current culture, which provides values used to govern formatting. Computers that use different settings generate different result strings.

In addition, if you use the [CultureInfo\(String\)](#) constructor to instantiate a new [CultureInfo](#) object that represents the same culture as the current system culture, any customizations established by the **Regional and Language Options** item in Control Panel will be applied to the new [CultureInfo](#) object. You can use the [CultureInfo\(String, Boolean\)](#) constructor to create a [CultureInfo](#) object that doesn't reflect a system's customizations.

DateTimeFormatInfo properties

Formatting is influenced by properties of the current [DateTimeFormatInfo](#) object, which is provided implicitly by the current culture or explicitly by the [IFormatProvider](#) parameter of the method that invokes formatting. For the [IFormatProvider](#) parameter,

you should specify a [CultureInfo](#) object, which represents a culture, or a [DateTimeFormatInfo](#) object.

The result string produced by many of the custom date and time format specifiers also depends on properties of the current [DateTimeFormatInfo](#) object. Your application can change the result produced by some custom date and time format specifiers by changing the corresponding [DateTimeFormatInfo](#) property. For example, the "ddd" format specifier adds an abbreviated weekday name found in the [AbbreviatedDayNames](#) string array to the result string. Similarly, the "MMMM" format specifier adds a full month name found in the [MonthNames](#) string array to the result string.

See also

- [System.DateTime](#)
- [System.IFormatProvider](#)
- [Formatting types](#)
- [Standard Date and Time format strings](#)
- [Sample: .NET Core WinForms formatting utility \(C#\)](#)
- [Sample: .NET Core WinForms formatting utility \(Visual Basic\)](#)

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

 .NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

Standard TimeSpan format strings

Article • 07/20/2022

A standard [TimeSpan](#) format string uses a single format specifier to define the text representation of a [TimeSpan](#) value that results from a formatting operation. Any format string that contains more than one character, including white space, is interpreted as a custom [TimeSpan](#) format string. For more information, see [Custom TimeSpan format strings](#).

The string representations of [TimeSpan](#) values are produced by calls to the overloads of the [TimeSpan.ToString](#) method, as well as by methods that support composite formatting, such as [String.Format](#). For more information, see [Formatting Types](#) and [Composite Formatting](#). The following example illustrates the use of standard format strings in formatting operations.

C#

```
using System;

public class Example
{
    public static void Main()
    {
        TimeSpan duration = new TimeSpan(1, 12, 23, 62);
        string output = "Time of Travel: " + duration.ToString("c");
        Console.WriteLine(output);

        Console.WriteLine("Time of Travel: {0:c}", duration);
    }
}
// The example displays the following output:
//      Time of Travel: 1.12:24:02
//      Time of Travel: 1.12:24:02
```

Standard [TimeSpan](#) format strings are also used by the [TimeSpan.ParseExact](#) and [TimeSpan.TryParseExact](#) methods to define the required format of input strings for parsing operations. (Parsing converts the string representation of a value to that value.) The following example illustrates the use of standard format strings in parsing operations.

C#

```
using System;

public class Example
{
```

```

public static void Main()
{
    string value = "1.03:14:56.1667";
    TimeSpan interval;
    try {
        interval = TimeSpan.ParseExact(value, "c", null);
        Console.WriteLine("Converted '{0}' to {1}", value, interval);
    }
    catch (FormatException) {
        Console.WriteLine("{0}: Bad Format", value);
    }
    catch (OverflowException) {
        Console.WriteLine("{0}: Out of Range", value);
    }

    if (TimeSpan.TryParseExact(value, "c", null, out interval))
        Console.WriteLine("Converted '{0}' to {1}", value, interval);
    else
        Console.WriteLine("Unable to convert {0} to a time interval.",
                          value);
}
// The example displays the following output:
//      Converted '1.03:14:56.1667' to 1.03:14:56.1667000
//      Converted '1.03:14:56.1667' to 1.03:14:56.1667000

```

The following table lists the standard time interval format specifiers.

| Format specifier | Name | Description | Examples |
|------------------|-----------------------------|--|--|
| "c" | Constant (invariant) format | This specifier is not culture-sensitive. It takes the form [-] [d'.']hh'':'mm'':'ss['.'fffffff]. (The "t" and "T" format strings produce the same results.) More information: The Constant ("c") Format Specifier . | TimeSpan.Zero -> 00:00:00 New TimeSpan(0, 0, 30, 0) -> 00:30:00 New TimeSpan(3, 17, 25, 30, 500) -> 3.17:25:30.5000000 |
| "g" | General short format | This specifier outputs only what is needed. It is culture-sensitive and takes the form [-] [d' ':']h' ':'mm' ':'ss[.FFFFFF]. More information: The General Short ("g") Format Specifier . | New TimeSpan(1, 3, 16, 50, 500) -> 1:3:16:50.5 (en-US) New TimeSpan(1, 3, 16, 50, 500) -> 1:3:16:50,5 (fr-FR) New TimeSpan(1, 3, 16, 50, 599) -> |

| Format specifier | Name | Description | Examples |
|------------------|---------------------|---|--|
| | | | 1:3:16:50.599 (en-US) |
| "G" | General long format | This specifier always outputs days and seven fractional digits. It is culture-sensitive and takes the form <code>[-]d':'hh':'mm':'ss.fffffff.</code> | New TimeSpan(1, 3, 16, 50, 599) -> 1:3:16:50,599 (fr-FR) More information: The General Long ("G") Format Specifier . |

The Constant ("c") Format Specifier

The "c" format specifier returns the string representation of a [TimeSpan](#) value in the following form:

`[-][d.]hh:mm:ss[.fffffff]`

Elements in square brackets ([and]) are optional. The period (.) and colon (:) are literal symbols. The following table describes the remaining elements.

| Element | Description |
|---------|--|
| - | An optional negative sign, which indicates a negative time interval. |
| d | The optional number of days, with no leading zeros. |
| hh | The number of hours, which ranges from "00" to "23". |
| mm | The number of minutes, which ranges from "00" to "59". |
| ss | The number of seconds, which ranges from "0" to "59". |
| fffffff | The optional fractional portion of a second. Its value can range from "0000001" (one tick, or one ten-millionth of a second) to "9999999" (9,999,999 ten-millionths of a second, or one second less one tick). |

Unlike the "g" and "G" format specifiers, the "c" format specifier is not culture-sensitive. It produces the string representation of a [TimeSpan](#) value that is invariant and that's

common to versions prior to .NET Framework 4. "c" is the default [TimeSpan](#) format string; the [TimeSpan.ToString\(\)](#) method formats a time interval value by using the "c" format string.

ⓘ Note

[TimeSpan](#) also supports the "t" and "T" standard format strings, which are identical in behavior to the "c" standard format string.

The following example instantiates two [TimeSpan](#) objects, uses them to perform arithmetic operations, and displays the result. In each case, it uses composite formatting to display the [TimeSpan](#) value by using the "c" format specifier.

C#

```
using System;

public class Example
{
    public static void Main()
    {
        TimeSpan interval1, interval2;
        interval1 = new TimeSpan(7, 45, 16);
        interval2 = new TimeSpan(18, 12, 38);

        Console.WriteLine("{0:c} - {1:c} = {2:c}", interval1,
                          interval2, interval1 - interval2);
        Console.WriteLine("{0:c} + {1:c} = {2:c}", interval1,
                          interval2, interval1 + interval2);

        interval1 = new TimeSpan(0, 0, 1, 14, 365);
        interval2 = TimeSpan.FromTicks(2143756);
        Console.WriteLine("{0:c} + {1:c} = {2:c}", interval1,
                          interval2, interval1 + interval2);
    }
}

// The example displays the following output:
//      07:45:16 - 18:12:38 = -10:27:22
//      07:45:16 + 18:12:38 = 1.01:57:54
//      00:01:14.3650000 + 00:00:00.2143756 = 00:01:14.5793756
```

The General Short ("g") Format Specifier

The "g" [TimeSpan](#) format specifier returns the string representation of a [TimeSpan](#) value in a compact form by including only the elements that are necessary. It has the following form:

[-][d:]h:mm:ss[.FFFFFF]

Elements in square brackets ([and]) are optional. The colon (:) is a literal symbol. The following table describes the remaining elements.

| Element | Description |
|---------|---|
| - | An optional negative sign, which indicates a negative time interval. |
| d | The optional number of days, with no leading zeros. |
| h | The number of hours, which ranges from "0" to "23", with no leading zeros. |
| mm | The number of minutes, which ranges from "00" to "59". |
| ss | The number of seconds, which ranges from "00" to "59". |
| . | The fractional seconds separator. It is equivalent to the specified culture's NumberDecimalSeparator property without user overrides. |
| FFFFFFF | The fractional seconds. As few digits as possible are displayed. |

Like the "G" format specifier, the "g" format specifier is localized. Its fractional seconds separator is based on either the current culture or a specified culture's [NumberDecimalSeparator](#) property.

The following example instantiates two `TimeSpan` objects, uses them to perform arithmetic operations, and displays the result. In each case, it uses composite formatting to display the `TimeSpan` value by using the "g" format specifier. In addition, it formats the `TimeSpan` value by using the formatting conventions of the current system culture (which, in this case, is English - United States or en-US) and the French - France (fr-FR) culture.

C#

```

        interval2, interval1 + interval2));

interval1 = new TimeSpan(0, 0, 1, 14, 36);
interval2 = TimeSpan.FromTicks(2143756);
Console.WriteLine("{0:g} + {1:g} = {2:g}", interval1,
                  interval2, interval1 + interval2);
}
}
// The example displays the following output:
//      7:45:16 - 18:12:38 = -10:27:22
//      7:45:16 + 18:12:38 = 1:1:57:54
//      0:01:14.036 + 0:00:00.2143756 = 0:01:14.2503756

```

The General Long ("G") Format Specifier

The "G" [TimeSpan](#) format specifier returns the string representation of a [TimeSpan](#) value in a long form that always includes both days and fractional seconds. The string that results from the "G" standard format specifier has the following form:

`[-]d:hh:mm:ss.fffffff`

Elements in square brackets ([and]) are optional. The colon (:) is a literal symbol. The following table describes the remaining elements.

| Element | Description |
|---------|---|
| - | An optional negative sign, which indicates a negative time interval. |
| d | The number of days, with no leading zeros. |
| hh | The number of hours, which ranges from "00" to "23". |
| mm | The number of minutes, which ranges from "00" to "59". |
| ss | The number of seconds, which ranges from "00" to "59". |
| . | The fractional seconds separator. It is equivalent to the specified culture's NumberDecimalSeparator property without user overrides. |
| fffffff | The fractional seconds. |

Like the "G" format specifier, the "g" format specifier is localized. Its fractional seconds separator is based on either the current culture or a specified culture's [NumberDecimalSeparator](#) property.

The following example instantiates two [TimeSpan](#) objects, uses them to perform arithmetic operations, and displays the result. In each case, it uses composite formatting to display the [TimeSpan](#) value by using the "G" format specifier. In addition, it formats

the `TimeSpan` value by using the formatting conventions of the current system culture (which, in this case, is English - United States or en-US) and the French - France (fr-FR) culture.

C#

```
using System;
using System.Globalization;

public class Example
{
    public static void Main()
    {
        TimeSpan interval1, interval2;
        interval1 = new TimeSpan(7, 45, 16);
        interval2 = new TimeSpan(18, 12, 38);

        Console.WriteLine("{0:G} - {1:G} = {2:G}", interval1,
                           interval2, interval1 - interval2);
        Console.WriteLine(String.Format(new CultureInfo("fr-FR"),
                                         "{0:G} + {1:G} = {2:G}", interval1,
                                         interval2, interval1 + interval2));

        interval1 = new TimeSpan(0, 0, 1, 14, 36);
        interval2 = TimeSpan.FromTicks(2143756);
        Console.WriteLine("{0:G} + {1:G} = {2:G}", interval1,
                           interval2, interval1 + interval2);
    }
}
// The example displays the following output:
//      0:07:45:16.0000000 - 0:18:12:38.0000000 = -0:10:27:22.0000000
//      0:07:45:16,0000000 + 0:18:12:38,0000000 = 1:01:57:54,0000000
//      0:00:01:14.0360000 + 0:00:00:00.2143756 = 0:00:01:14.2503756
```

See also

- [Formatting Types](#)
- [Custom TimeSpan Format Strings](#)
- [Parsing Strings](#)

 Collaborate with us on
GitHub

The source for this content can
be found on GitHub, where you
can also create and review

.NET

.NET feedback

The .NET documentation is open
source. Provide feedback here.

 [Open a documentation issue](#)

issues and pull requests. For more information, see [our contributor guide](#).

 [Provide product feedback](#)

Custom TimeSpan format strings

Article • 09/15/2021

A [TimeSpan](#) format string defines the string representation of a [TimeSpan](#) value that results from a formatting operation. A custom format string consists of one or more custom [TimeSpan](#) format specifiers along with any number of literal characters. Any string that isn't a [Standard TimeSpan format string](#) is interpreted as a custom [TimeSpan](#) format string.

ⓘ Important

The custom [TimeSpan](#) format specifiers don't include placeholder separator symbols, such as the symbols that separate days from hours, hours from minutes, or seconds from fractional seconds. Instead, these symbols must be included in the custom format string as string literals. For example, "dd\.hh\:mm" defines a period (.) as the separator between days and hours, and a colon (:) as the separator between hours and minutes.

Custom [TimeSpan](#) format specifiers also don't include a sign symbol that enables you to differentiate between negative and positive time intervals. To include a sign symbol, you have to construct a format string by using conditional logic. The [Other characters](#) section includes an example.

The string representations of [TimeSpan](#) values are produced by calls to the overloads of the [TimeSpan.ToString](#) method, and by methods that support composite formatting, such as [String.Format](#). For more information, see [Formatting Types](#) and [Composite Formatting](#). The following example illustrates the use of custom format strings in formatting operations.

C#

```
using System;

public class Example
{
    public static void Main()
    {
        TimeSpan duration = new TimeSpan(1, 12, 23, 62);

        string output = null;
        output = "Time of Travel: " + duration.ToString("%d") + " days";
        Console.WriteLine(output);
        output = "Time of Travel: " + duration.ToString(@"dd\.hh\:mm\:ss"));
    }
}
```

```

        Console.WriteLine(output);

        Console.WriteLine("Time of Travel: {0:%d} day(s)", duration);
        Console.WriteLine("Time of Travel: {0:dd\\.hh\\:mm\\:ss} days",
duration);
    }
}
// The example displays the following output:
//      Time of Travel: 1 days
//      Time of Travel: 01.12:24:02
//      Time of Travel: 1 day(s)
//      Time of Travel: 01.12:24:02 days

```

Custom [TimeSpan](#) format strings are also used by the [TimeSpan.ParseExact](#) and [TimeSpan.TryParseExact](#) methods to define the required format of input strings for parsing operations. (Parsing converts the string representation of a value to that value.) The following example illustrates the use of standard format strings in parsing operations.

C#

```

using System;

public class Example
{
    public static void Main()
    {
        string value = null;
        TimeSpan interval;

        value = "6";
        if (TimeSpan.TryParseExact(value, "%d", null, out interval))
            Console.WriteLine("{0} --> {1}", value, interval.ToString("c"));
        else
            Console.WriteLine("Unable to parse '{0}'", value);

        value = "16:32.05";
        if (TimeSpan.TryParseExact(value, @"mm\:ss\.\ff", null, out interval))
            Console.WriteLine("{0} --> {1}", value, interval.ToString("c"));
        else
            Console.WriteLine("Unable to parse '{0}'", value);

        value= "12.035";
        if (TimeSpan.TryParseExact(value, "ss\.\fff", null, out interval))
            Console.WriteLine("{0} --> {1}", value, interval.ToString("c"));
        else
            Console.WriteLine("Unable to parse '{0}'", value);
    }
}
// The example displays the following output:
//      6 --> 6.00:00:00

```

```
//      16:32.05 --> 00:16:32.0500000
//      12.035 --> 00:00:12.0350000
```

The following table describes the custom date and time format specifiers.

| Format specifier | Description | Example |
|----------------------|--|---|
| "d", "%d" | The number of whole days in the time interval. More information: The "d" custom format specifier . | <code>new TimeSpan(6, 14, 32, 17, 685): %d --> "6"</code> <code>d\.\hh\:\mm --> "6.14:32"</code> |
| "dd"- "ddddddddd" | The number of whole days in the time interval, padded with leading zeros as needed. More information: The "dd"- "ddddddddd" custom format specifiers . | <code>new TimeSpan(6, 14, 32, 17, 685): ddd --> "006"</code> <code>dd\.\hh\:\mm --> "06.14:32"</code> |
| "h", "%h" | The number of whole hours in the time interval that aren't counted as part of days. Single-digit hours don't have a leading zero. More information: The "h" custom format specifier . | <code>new TimeSpan(6, 14, 32, 17, 685): %h --> "14"</code> <code>hh\:\mm --> "14:32"</code> |
| "hh" | The number of whole hours in the time interval that aren't counted as part of days. Single-digit hours have a leading zero. More information: The "hh" custom format specifier . | <code>new TimeSpan(6, 14, 32, 17, 685): hh --> "14"</code> <code>new TimeSpan(6, 8, 32, 17, 685): hh --> 08</code> |
| "m", "%m" | The number of whole minutes in the time interval that aren't included as part of hours or days. Single-digit minutes don't have a leading zero. More information: The "m" custom format specifier . | <code>new TimeSpan(6, 14, 8, 17, 685): %m --> "8"</code> <code>h\:\m --> "14:8"</code> |
| "mm" | The number of whole minutes in the time interval that aren't included as part of hours or days. Single-digit minutes have a leading zero. | <code>new TimeSpan(6, 14, 8, 17, 685): mm --> "08"</code> |

| Format specifier | Description | Example |
|------------------|--|---|
| | <p>More information: The "mm" custom format specifier.</p> | <pre>new TimeSpan(6, 8, 5, 17, 685):</pre> <pre>d\.\hh\:\mm\:\ss --> 6.08:05:17</pre> |
| 's', "%s" | <p>The number of whole seconds in the time interval that aren't included as part of hours, days, or minutes. Single-digit seconds don't have a leading zero.</p> <p>More information: The "s" custom format specifier.</p> | <pre>TimeSpan.FromSeconds(12.965):</pre> <pre>%s --> 12</pre> <pre>s\.\fff --> 12.965</pre> |
| 'ss' | <p>The number of whole seconds in the time interval that aren't included as part of hours, days, or minutes. Single-digit seconds have a leading zero.</p> <p>More information: The "ss" custom format specifier.</p> | <pre>TimeSpan.FromSeconds(6.965):</pre> <pre>ss --> 06</pre> <pre>ss\.\fff --> 06.965</pre> |
| 'f', "%f" | <p>The tenths of a second in a time interval.</p> <p>More information: The "f" custom format specifier.</p> | <pre>TimeSpan.FromSeconds(6.895):</pre> <pre>f --> 8</pre> <pre>ss\.\f --> 06.8</pre> |
| 'ff' | <p>The hundredths of a second in a time interval.</p> <p>More information: The "ff" custom format specifier.</p> | <pre>TimeSpan.FromSeconds(6.895):</pre> <pre>ff --> 89</pre> <pre>ss\.\ff --> 06.89</pre> |
| 'fff' | <p>The milliseconds in a time interval.</p> <p>More information: The "fff" custom format specifier.</p> | <pre>TimeSpan.FromSeconds(6.895):</pre> <pre>fff --> 895</pre> <pre>ss\.\fff --> 06.895</pre> |
| 'ffff' | <p>The ten-thousandths of a second in a time interval.</p> <p>More information: The "ffff" custom format specifier.</p> | <pre>TimeSpan.Parse("0:0:6.8954321"):</pre> <pre>ffff --> 8954</pre> <pre>ss\.\ffff --> 06.8954</pre> |
| 'ffffff' | <p>The hundred-thousandths of a second in a time interval.</p> | <pre>TimeSpan.Parse("0:0:6.8954321"):</pre> <pre>ffffff --> 89543</pre> |

| Format specifier | Description | Example |
|------------------|--|--|
| | More information: The "fffff" custom format specifier . | <code>ss\.\fffff</code> --> 06.89543 |
| 'fffff' | The millionths of a second in a time interval. More information: The "fffff" custom format specifier . | <code>TimeSpan.Parse("0:0:6.8954321"):</code> <code>fffff</code> --> 895432 <code>ss\.\fffff</code> --> 06.895432 |
| 'fffffff' | The ten-millionths of a second (or the fractional ticks) in a time interval. More information: The "fffffff" custom format specifier . | <code>TimeSpan.Parse("0:0:6.8954321"):</code> <code>fffffff</code> --> 8954321 <code>ss\.\fffffff</code> --> 06.8954321 |
| 'F', "%F" | The tenths of a second in a time interval. Nothing is displayed if the digit is zero. More information: The "F" custom format specifier . | <code>TimeSpan.Parse("00:00:06.32"):</code> <code>%F</code> : 3 <code>TimeSpan.Parse("0:0:3.091"):</code> <code>ss\.\F</code> : 03. |
| 'FF' | The hundredths of a second in a time interval. Any fractional trailing zeros or two zero digits aren't included. More information: The "FF" custom format specifier . | <code>TimeSpan.Parse("00:00:06.329"):</code> <code>FF</code> : 32 <code>TimeSpan.Parse("0:0:3.101"):</code> <code>ss\.\FF</code> : 03.1 |
| 'FFF' | The milliseconds in a time interval. Any fractional trailing zeros aren't included. More information: | <code>TimeSpan.Parse("00:00:06.3291"):</code> <code>FFF</code> : 329 <code>TimeSpan.Parse("0:0:3.1009"):</code> <code>ss\.\FFF</code> : 03.1 |
| 'FFFF' | The ten-thousandths of a second in a time interval. Any fractional trailing zeros aren't included. More information: The "FFFF" custom format specifier . | <code>TimeSpan.Parse("00:00:06.32917"):</code> <code>FFFF</code> : 3291 <code>TimeSpan.Parse("0:0:3.10009"):</code> <code>ss\.\FFFF</code> : 03.1 |

| Format specifier | Description | Example |
|---------------------|--|--|
| "FFFFF" | The hundred-thousandths of a second in a time interval. Any fractional trailing zeros aren't included. More information: The "FFFFF" custom format specifier . | <code>TimeSpan.Parse("00:00:06.329179"):</code> <code>FFFFF : 32917</code> <code>TimeSpan.Parse("0:0:3.100009"):</code> <code>ss\.\FFFFF : 03.1</code> |
| "FFFFFF" | The millionths of a second in a time interval. Any fractional trailing zeros aren't displayed. More information: The "FFFFFF" custom format specifier . | <code>TimeSpan.Parse("00:00:06.3291791"):</code> <code>FFFFFF : 329179</code> <code>TimeSpan.Parse("0:0:3.1000009"):</code> <code>ss\.\FFFFFF : 03.1</code> |
| "FFFFFFF" | The ten-millionths of a second in a time interval. Any fractional trailing zeros or seven zeros aren't displayed. More information: The "FFFFFFF" custom format specifier . | <code>TimeSpan.Parse("00:00:06.3291791"):</code> <code>FFFFFFF : 3291791</code> <code>TimeSpan.Parse("0:0:3.1900000"):</code> <code>ss\.\FFFFFFF : 03.19</code> |
| 'string' | Literal string delimiter. More information: Other characters . | <code>new TimeSpan(14, 32, 17):</code> <code>hh' : 'mm' : 'ss --> "14:32:17"</code> |
| \ | The escape character. More information: Other characters . | <code>new TimeSpan(14, 32, 17):</code> <code>hh\ :mm\ :ss --> "14:32:17"</code> |
| Any other character | Any other unescaped character is interpreted as a custom format specifier. More Information: Other characters . | <code>new TimeSpan(14, 32, 17):</code> <code>hh\ :mm\ :ss --> "14:32:17"</code> |

The "d" custom format specifier

The "d" custom format specifier outputs the value of the `TimeSpan.Days` property, which represents the number of whole days in the time interval. It outputs the full number of days in a `TimeSpan` value, even if the value has more than one digit. If the value of the `TimeSpan.Days` property is zero, the specifier outputs "0".

If the "d" custom format specifier is used alone, specify "%d" so that it isn't misinterpreted as a standard format string. The following example provides an illustration.

```
C#
```

```
TimeSpan ts1 = new TimeSpan(16, 4, 3, 17, 250);
Console.WriteLine(ts1.ToString("%d"));
// Displays 16
```

The following example illustrates the use of the "d" custom format specifier.

```
C#
```

```
TimeSpan ts2 = new TimeSpan(4, 3, 17);
Console.WriteLine(ts2.ToString(@"d\.\hh\:mm\:ss"));

TimeSpan ts3 = new TimeSpan(3, 4, 3, 17);
Console.WriteLine(ts3.ToString(@"d\.\hh\:mm\:ss"));
// The example displays the following output:
//      0.04:03:17
//      3.04:03:17
```

[Back to table](#)

The "dd"- "dddddd" custom format specifiers

The "dd", "ddd", "dddd", "ddddd", "ddyyyy", "ddyyyyy", and "ddyyyyyy" custom format specifiers output the value of the [TimeSpan.Days](#) property, which represents the number of whole days in the time interval.

The output string includes a minimum number of digits specified by the number of "d" characters in the format specifier, and it's padded with leading zeros as needed. If the digits in the number of days exceed the number of "d" characters in the format specifier, the full number of days is output in the result string.

The following example uses these format specifiers to display the string representation of two [TimeSpan](#) values. The value of the days component of the first time interval is zero; the value of the days component of the second is 365.

```
C#
```

```
TimeSpan ts1 = new TimeSpan(0, 23, 17, 47);
TimeSpan ts2 = new TimeSpan(365, 21, 19, 45);
```

```

for (int ctr = 2; ctr <= 8; ctr++)
{
    string fmt = new String('d', ctr) + @"\.\hh\:\mm\:\ss";
    Console.WriteLine("{0} --> {1:" + fmt + "}", fmt, ts1);
    Console.WriteLine("{0} --> {1:" + fmt + "}", fmt, ts2);
    Console.WriteLine();
}
// The example displays the following output:
//      dd\.hh\:mm\:ss --> 00.23:17:47
//      dd\.hh\:mm\:ss --> 365.21:19:45
//
//      ddd\.hh\:mm\:ss --> 000.23:17:47
//      ddd\.hh\:mm\:ss --> 365.21:19:45
//
//      dddd\.hh\:mm\:ss --> 0000.23:17:47
//      dddd\.hh\:mm\:ss --> 00365.21:19:45
//
//      dddddd\.hh\:mm\:ss --> 000000.23:17:47
//      dddddd\.hh\:mm\:ss --> 000365.21:19:45
//
//      ddddddd\.hh\:mm\:ss --> 0000000.23:17:47
//      ddddddd\.hh\:mm\:ss --> 0000365.21:19:45

```

[Back to table](#)

The "h" custom format specifier

The "h" custom format specifier outputs the value of the `TimeSpan.Hours` property, which represents the number of whole hours in the time interval that isn't counted as part of its day component. It returns a one-digit string value if the value of the `TimeSpan.Hours` property is 0 through 9, and it returns a two-digit string value if the value of the `TimeSpan.Hours` property ranges from 10 to 23.

If the "h" custom format specifier is used alone, specify "%h" so that it isn't misinterpreted as a standard format string. The following example provides an illustration.

C#

```

TimeSpan ts = new TimeSpan(3, 42, 0);
Console.WriteLine("{0:%h} hours {0:%m} minutes", ts);

```

```
// The example displays the following output:  
//      3 hours 42 minutes
```

Ordinarily, in a parsing operation, an input string that includes only a single number is interpreted as the number of days. You can use the "%h" custom format specifier instead to interpret the numeric string as the number of hours. The following example provides an illustration.

C#

```
string value = "8";  
TimeSpan interval;  
if (TimeSpan.TryParseExact(value, "%h", null, out interval))  
    Console.WriteLine(interval.ToString("c"));  
else  
    Console.WriteLine("Unable to convert '{0}' to a time interval",  
                      value);  
// The example displays the following output:  
//      08:00:00
```

The following example illustrates the use of the "h" custom format specifier.

C#

```
TimeSpan ts1 = new TimeSpan(14, 3, 17);  
Console.WriteLine(ts1.ToString(@"d\.h\:mm\:ss"));  
  
TimeSpan ts2 = new TimeSpan(3, 4, 3, 17);  
Console.WriteLine(ts2.ToString(@"d\.h\:mm\:ss"));  
// The example displays the following output:  
//      0.14:03:17  
//      3.4:03:17
```

[Back to table](#)

The "hh" custom format specifier

The "hh" custom format specifier outputs the value of the [TimeSpan.Hours](#) property, which represents the number of whole hours in the time interval that isn't counted as part of its day component. For values from 0 through 9, the output string includes a leading zero.

Ordinarily, in a parsing operation, an input string that includes only a single number is interpreted as the number of days. You can use the "hh" custom format specifier instead

to interpret the numeric string as the number of hours. The following example provides an illustration.

```
C#
```

```
string value = "08";
TimeSpan interval;
if (TimeSpan.TryParseExact(value, "hh", null, out interval))
    Console.WriteLine(interval.ToString("c"));
else
    Console.WriteLine("Unable to convert '{0}' to a time interval",
                     value);
// The example displays the following output:
//      08:00:00
```

The following example illustrates the use of the "hh" custom format specifier.

```
C#
```

```
TimeSpan ts1 = new TimeSpan(14, 3, 17);
Console.WriteLine(ts1.ToString(@"d\.\hh\:mm\:ss"));

TimeSpan ts2 = new TimeSpan(3, 4, 3, 17);
Console.WriteLine(ts2.ToString(@"d\.\hh\:mm\:ss"));
// The example displays the following output:
//      0.14:03:17
//      3.04:03:17
```

[Back to table](#)

The "m" custom format specifier

The "m" custom format specifier outputs the value of the [TimeSpan.Minutes](#) property, which represents the number of whole minutes in the time interval that isn't counted as part of its day component. It returns a one-digit string value if the value of the [TimeSpan.Minutes](#) property is 0 through 9, and it returns a two-digit string value if the value of the [TimeSpan.Minutes](#) property ranges from 10 to 59.

If the "m" custom format specifier is used alone, specify "%m" so that it isn't misinterpreted as a standard format string. The following example provides an illustration.

```
C#
```

```
TimeSpan ts = new TimeSpan(3, 42, 0);
Console.WriteLine("{0:%h} hours {0:%m} minutes", ts);
```

```
// The example displays the following output:  
//      3 hours 42 minutes
```

Ordinarily, in a parsing operation, an input string that includes only a single number is interpreted as the number of days. You can use the "%m" custom format specifier instead to interpret the numeric string as the number of minutes. The following example provides an illustration.

C#

```
string value = "3";  
TimeSpan interval;  
if (TimeSpan.TryParseExact(value, "%m", null, out interval))  
    Console.WriteLine(interval.ToString("c"));  
else  
    Console.WriteLine("Unable to convert '{0}' to a time interval",  
                      value);  
// The example displays the following output:  
//      00:03:00
```

The following example illustrates the use of the "m" custom format specifier.

C#

```
TimeSpan ts1 = new TimeSpan(0, 6, 32);  
Console.WriteLine("{0:m\\:ss} minutes", ts1);  
  
TimeSpan ts2 = new TimeSpan(3, 4, 3, 17);  
Console.WriteLine("Elapsed time: {0:m\\:ss}", ts2);  
// The example displays the following output:  
//      6:32 minutes  
//      Elapsed time: 18:44
```

[Back to table](#)

The "mm" custom format specifier

The "mm" custom format specifier outputs the value of the [TimeSpan.Minutes](#) property, which represents the number of whole minutes in the time interval that isn't included as part of its hours or days component. For values from 0 through 9, the output string includes a leading zero.

Ordinarily, in a parsing operation, an input string that includes only a single number is interpreted as the number of days. You can use the "mm" custom format specifier

instead to interpret the numeric string as the number of minutes. The following example provides an illustration.

C#

```
string value = "07";
TimeSpan interval;
if (TimeSpan.TryParseExact(value, "mm", null, out interval))
    Console.WriteLine(interval.ToString("c"));
else
    Console.WriteLine("Unable to convert '{0}' to a time interval",
                      value);
// The example displays the following output:
//      00:07:00
```

The following example illustrates the use of the "mm" custom format specifier.

C#

```
TimeSpan departTime = new TimeSpan(11, 12, 00);
TimeSpan arriveTime = new TimeSpan(16, 28, 00);
Console.WriteLine("Travel time: {0:hh\\:\\mm}",
                  arriveTime - departTime);
// The example displays the following output:
//      Travel time: 05:16
```

[Back to table](#)

The "s" custom format specifier

The "s" custom format specifier outputs the value of the `TimeSpan.Seconds` property, which represents the number of whole seconds in the time interval that isn't included as part of its hours, days, or minutes component. It returns a one-digit string value if the value of the `TimeSpan.Seconds` property is 0 through 9, and it returns a two-digit string value if the value of the `TimeSpan.Seconds` property ranges from 10 to 59.

If the "s" custom format specifier is used alone, specify "%s" so that it isn't misinterpreted as a standard format string. The following example provides an illustration.

C#

```
TimeSpan ts = TimeSpan.FromSeconds(12.465);
Console.WriteLine(ts.ToString("%s"));
```

```
// The example displays the following output:  
//      12
```

Ordinarily, in a parsing operation, an input string that includes only a single number is interpreted as the number of days. You can use the "%s" custom format specifier instead to interpret the numeric string as the number of seconds. The following example provides an illustration.

C#

```
string value = "9";  
TimeSpan interval;  
if (TimeSpan.TryParseExact(value, "%s", null, out interval))  
    Console.WriteLine(interval.ToString("c"));  
else  
    Console.WriteLine("Unable to convert '{0}' to a time interval",  
                      value);  
// The example displays the following output:  
//      00:00:09
```

The following example illustrates the use of the "s" custom format specifier.

C#

```
TimeSpan startTime = new TimeSpan(0, 12, 30, 15, 0);  
TimeSpan endTime = new TimeSpan(0, 12, 30, 21, 3);  
Console.WriteLine(@"Elapsed Time: {0:s\:fff} seconds",  
                 endTime - startTime);  
// The example displays the following output:  
//      Elapsed Time: 6:003 seconds
```

[Back to table](#)

The "ss" custom format specifier

The "ss" custom format specifier outputs the value of the [TimeSpan.Seconds](#) property, which represents the number of whole seconds in the time interval that isn't included as part of its hours, days, or minutes component. For values from 0 through 9, the output string includes a leading zero.

Ordinarily, in a parsing operation, an input string that includes only a single number is interpreted as the number of days. You can use the "ss" custom format specifier instead to interpret the numeric string as the number of seconds. The following example provides an illustration.

C#

```
string[] values = { "49", "9", "06" };
TimeSpan interval;
foreach (string value in values)
{
    if (TimeSpan.TryParseExact(value, "ss", null, out interval))
        Console.WriteLine(interval.ToString("c"));
    else
        Console.WriteLine("Unable to convert '{0}' to a time interval",
                         value);
}
// The example displays the following output:
//      00:00:49
//      Unable to convert '9' to a time interval
//      00:00:06
```

The following example illustrates the use of the "ss" custom format specifier.

C#

```
TimeSpan interval1 = TimeSpan.FromSeconds(12.60);
Console.WriteLine(interval1.ToString(@"ss\.\fff"));

TimeSpan interval2 = TimeSpan.FromSeconds(6.485);
Console.WriteLine(interval2.ToString(@"ss\.\fff"));
// The example displays the following output:
//      12.600
//      06.485
```

[Back to table](#)

The "f" custom format specifier

The "f" custom format specifier outputs the tenths of a second in a time interval. In a formatting operation, any remaining fractional digits are truncated. In a parsing operation that calls the [TimeSpan.ParseExact](#) or [TimeSpan.TryParseExact](#) method, the input string must contain exactly one fractional digit.

If the "f" custom format specifier is used alone, specify "%f" so that it isn't misinterpreted as a standard format string.

The following example uses the "f" custom format specifier to display the tenths of a second in a [TimeSpan](#) value. "f" is used first as the only format specifier, and then combined with the "s" specifier in a custom format string.

C#

```

TimeSpan ts = new TimeSpan(1003498765432);
string fmt;
Console.WriteLine(ts.ToString("c"));
Console.WriteLine();

for (int ctr = 1; ctr <= 7; ctr++) {
    fmt = new String('f', ctr);
    if (fmt.Length == 1) fmt = "%" + fmt;
    Console.WriteLine("{0,10}: {1:" + fmt + "}", fmt, ts);
}
Console.WriteLine();

for (int ctr = 1; ctr <= 7; ctr++) {
    fmt = new String('f', ctr);
    Console.WriteLine("{0,10}: {1:s\\.{" + fmt + "}}", "s\\.{" + fmt, ts);
}
// The example displays the following output:
//          %f: 8
//          ff: 87
//          fff: 876
//          ffff: 8765
//          fffff: 87654
//          ffffff: 876543
//          ffffffff: 8765432
//
//          s\.f: 29.8
//          s\.ff: 29.87
//          s\.fff: 29.876
//          s\.ffff: 29.8765
//          s\.fffff: 29.87654
//          s\.fffffff: 29.876543
//          s\.fffffff: 29.8765432

```

[Back to table](#)

The "ff" custom format specifier

The "ff" custom format specifier outputs the hundredths of a second in a time interval. In a formatting operation, any remaining fractional digits are truncated. In a parsing operation that calls the [TimeSpan.ParseExact](#) or [TimeSpan.TryParseExact](#) method, the input string must contain exactly two fractional digits.

The following example uses the "ff" custom format specifier to display the hundredths of a second in a [TimeSpan](#) value. "ff" is used first as the only format specifier, and then combined with the "s" specifier in a custom format string.

C#

```

TimeSpan ts = new TimeSpan(1003498765432);
string fmt;
Console.WriteLine(ts.ToString("c"));
Console.WriteLine();

for (int ctr = 1; ctr <= 7; ctr++) {
    fmt = new String('f', ctr);
    if (fmt.Length == 1) fmt = "%" + fmt;
    Console.WriteLine("{0,10}: {1:" + fmt + "}", fmt, ts);
}
Console.WriteLine();

for (int ctr = 1; ctr <= 7; ctr++) {
    fmt = new String('f', ctr);
    Console.WriteLine("{0,10}: {1:s\\.{" + fmt + "}}", "s\\.{" + fmt, ts);
}
// The example displays the following output:
//          %f: 8
//          ff: 87
//          fff: 876
//          ffff: 8765
//          fffff: 87654
//          ffffff: 876543
//          ffffffff: 8765432
//
//          s\.f: 29.8
//          s\.ff: 29.87
//          s\.fff: 29.876
//          s\.ffff: 29.8765
//          s\.fffff: 29.87654
//          s\.fffffff: 29.876543
//          s\.fffffff: 29.8765432

```

[Back to table](#)

The "fff" custom format specifier

The "fff" custom format specifier (with three "f" characters) outputs the milliseconds in a time interval. In a formatting operation, any remaining fractional digits are truncated. In a parsing operation that calls the [TimeSpan.ParseExact](#) or [TimeSpan.TryParseExact](#) method, the input string must contain exactly three fractional digits.

The following example uses the "fff" custom format specifier to display the milliseconds in a [TimeSpan](#) value. "fff" is used first as the only format specifier, and then combined with the "s" specifier in a custom format string.

C#

```

TimeSpan ts = new TimeSpan(1003498765432);
string fmt;
Console.WriteLine(ts.ToString("c"));
Console.WriteLine();

for (int ctr = 1; ctr <= 7; ctr++) {
    fmt = new String('f', ctr);
    if (fmt.Length == 1) fmt = "%" + fmt;
    Console.WriteLine("{0,10}: {1:" + fmt + "}", fmt, ts);
}
Console.WriteLine();

for (int ctr = 1; ctr <= 7; ctr++) {
    fmt = new String('f', ctr);
    Console.WriteLine("{0,10}: {1:s\\.{" + fmt + "}}", "s\\.{" + fmt, ts);
}
// The example displays the following output:
//          %f: 8
//          ff: 87
//          fff: 876
//          ffff: 8765
//          fffff: 87654
//          ffffff: 876543
//          ffffffff: 8765432
//
//          s\.f: 29.8
//          s\.ff: 29.87
//          s\.fff: 29.876
//          s\.ffff: 29.8765
//          s\.fffff: 29.87654
//          s\.fffffff: 29.876543
//          s\.fffffff: 29.8765432

```

[Back to table](#)

The "ffff" custom format specifier

The "ffff" custom format specifier (with four "f" characters) outputs the ten-thousandths of a second in a time interval. In a formatting operation, any remaining fractional digits are truncated. In a parsing operation that calls the [TimeSpan.ParseExact](#) or [TimeSpan.TryParseExact](#) method, the input string must contain exactly four fractional digits.

The following example uses the "ffff" custom format specifier to display the ten-thousandths of a second in a [TimeSpan](#) value. "ffff" is used first as the only format specifier, and then combined with the "s" specifier in a custom format string.

C#

```

TimeSpan ts = new TimeSpan(1003498765432);
string fmt;
Console.WriteLine(ts.ToString("c"));
Console.WriteLine();

for (int ctr = 1; ctr <= 7; ctr++) {
    fmt = new String('f', ctr);
    if (fmt.Length == 1) fmt = "%" + fmt;
    Console.WriteLine("{0,10}: {1:" + fmt + "}", fmt, ts);
}
Console.WriteLine();

for (int ctr = 1; ctr <= 7; ctr++) {
    fmt = new String('f', ctr);
    Console.WriteLine("{0,10}: {1:s\\.{" + fmt + "}}", "s\\.{" + fmt, ts);
}
// The example displays the following output:
//          %f: 8
//          ff: 87
//          fff: 876
//          ffff: 8765
//          fffff: 87654
//          ffffff: 876543
//          ffffffff: 8765432
//
//          s\.f: 29.8
//          s\.ff: 29.87
//          s\.fff: 29.876
//          s\.ffff: 29.8765
//          s\.fffff: 29.87654
//          s\.fffffff: 29.876543
//          s\.fffffff: 29.8765432

```

[Back to table](#)

The "fffff" custom format specifier

The "fffff" custom format specifier (with five "f" characters) outputs the hundred-thousandths of a second in a time interval. In a formatting operation, any remaining fractional digits are truncated. In a parsing operation that calls the [TimeSpan.ParseExact](#) or [TimeSpan.TryParseExact](#) method, the input string must contain exactly five fractional digits.

The following example uses the "fffff" custom format specifier to display the hundred-thousandths of a second in a [TimeSpan](#) value. "fffff" is used first as the only format specifier, and then combined with the "s" specifier in a custom format string.

C#

```

TimeSpan ts = new TimeSpan(1003498765432);
string fmt;
Console.WriteLine(ts.ToString("c"));
Console.WriteLine();

for (int ctr = 1; ctr <= 7; ctr++) {
    fmt = new String('f', ctr);
    if (fmt.Length == 1) fmt = "%" + fmt;
    Console.WriteLine("{0,10}: {1:" + fmt + "}", fmt, ts);
}
Console.WriteLine();

for (int ctr = 1; ctr <= 7; ctr++) {
    fmt = new String('f', ctr);
    Console.WriteLine("{0,10}: {1:s\\.{" + fmt + "}}", "s\\.{" + fmt, ts);
}
// The example displays the following output:
//          %f: 8
//          ff: 87
//          fff: 876
//          ffff: 8765
//          fffff: 87654
//          ffffff: 876543
//          ffffffff: 8765432
//
//          s\.f: 29.8
//          s\.ff: 29.87
//          s\.fff: 29.876
//          s\.ffff: 29.8765
//          s\.fffff: 29.87654
//          s\.fffffff: 29.876543
//          s\.fffffff: 29.8765432

```

[Back to table](#)

The "fffffff" custom format specifier

The "fffffff" custom format specifier (with six "f" characters) outputs the millionths of a second in a time interval. In a formatting operation, any remaining fractional digits are truncated. In a parsing operation that calls the [TimeSpan.ParseExact](#) or [TimeSpan.TryParseExact](#) method, the input string must contain exactly six fractional digits.

The following example uses the "fffffff" custom format specifier to display the millionths of a second in a [TimeSpan](#) value. It is used first as the only format specifier, and then combined with the "s" specifier in a custom format string.

C#

```

TimeSpan ts = new TimeSpan(1003498765432);
string fmt;
Console.WriteLine(ts.ToString("c"));
Console.WriteLine();

for (int ctr = 1; ctr <= 7; ctr++) {
    fmt = new String('f', ctr);
    if (fmt.Length == 1) fmt = "%" + fmt;
    Console.WriteLine("{0,10}: {1:" + fmt + "}", fmt, ts);
}
Console.WriteLine();

for (int ctr = 1; ctr <= 7; ctr++) {
    fmt = new String('f', ctr);
    Console.WriteLine("{0,10}: {1:s\\.{" + fmt + "}}", "s\\.{" + fmt, ts);
}
// The example displays the following output:
//          %f: 8
//          ff: 87
//          fff: 876
//          ffff: 8765
//          fffff: 87654
//          ffffff: 876543
//          ffffffff: 8765432
//
//          s\.f: 29.8
//          s\.ff: 29.87
//          s\.fff: 29.876
//          s\.ffff: 29.8765
//          s\.fffff: 29.87654
//          s\.fffffff: 29.876543
//          s\.fffffff: 29.8765432

```

[Back to table](#)

The "fffffff" custom format specifier

The "fffffff" custom format specifier (with seven "f" characters) outputs the ten-millionths of a second (or the fractional number of ticks) in a time interval. In a parsing operation that calls the `TimeSpan.ParseExact` or `TimeSpan.TryParseExact` method, the input string must contain exactly seven fractional digits.

The following example uses the "fffffff" custom format specifier to display the fractional number of ticks in a `TimeSpan` value. It is used first as the only format specifier, and then combined with the "s" specifier in a custom format string.

C#

```

TimeSpan ts = new TimeSpan(1003498765432);
string fmt;
Console.WriteLine(ts.ToString("c"));
Console.WriteLine();

for (int ctr = 1; ctr <= 7; ctr++) {
    fmt = new String('f', ctr);
    if (fmt.Length == 1) fmt = "%" + fmt;
    Console.WriteLine("{0,10}: {1:" + fmt + "}", fmt, ts);
}
Console.WriteLine();

for (int ctr = 1; ctr <= 7; ctr++) {
    fmt = new String('f', ctr);
    Console.WriteLine("{0,10}: {1:s\\.{" + fmt + "}}", "s\\.{" + fmt, ts);
}
// The example displays the following output:
//          %f: 8
//          ff: 87
//          fff: 876
//          ffff: 8765
//          fffff: 87654
//          ffffff: 876543
//          ffffffff: 8765432
//
//          s\.f: 29.8
//          s\.ff: 29.87
//          s\.fff: 29.876
//          s\.ffff: 29.8765
//          s\.fffff: 29.87654
//          s\.fffffff: 29.876543
//          s\.fffffff: 29.8765432

```

[Back to table](#)

The "F" custom format specifier

The "F" custom format specifier outputs the tenths of a second in a time interval. In a formatting operation, any remaining fractional digits are truncated. If the value of the time interval's tenths of a second is zero, it isn't included in the result string. In a parsing operation that calls the [TimeSpan.ParseExact](#) or [TimeSpan.TryParseExact](#) method, the presence of the tenths of a second digit is optional.

If the "F" custom format specifier is used alone, specify "%F" so that it isn't misinterpreted as a standard format string.

The following example uses the "F" custom format specifier to display the tenths of a second in a [TimeSpan](#) value. It also uses this custom format specifier in a parsing

operation.

C#

```
Console.WriteLine("Formatting:");
TimeSpan ts1 = TimeSpan.Parse("0:0:3.669");
Console.WriteLine("{0} ('%F') --> {0:%F}", ts1);

TimeSpan ts2 = TimeSpan.Parse("0:0:3.091");
Console.WriteLine("{0} ('ss\\.F') --> {0:ss\\.F}", ts2);
Console.WriteLine();

Console.WriteLine("Parsing:");
string[] inputs = { "0:0:03.", "0:0:03.1", "0:0:03.12" };
string fmt = @"h\:m\:ss\.F";
TimeSpan ts3;

foreach (string input in inputs) {
    if (TimeSpan.TryParseExact(input, fmt, null, out ts3))
        Console.WriteLine("{0} ('{1}') --> {2}", input, fmt, ts3);
    else
        Console.WriteLine("Cannot parse {0} with '{1}'.",
                         input, fmt);
}
// The example displays the following output:
//      Formatting:
//      00:00:03.6690000 ('%F') --> 6
//      00:00:03.0910000 ('ss\.F') --> 03.

//
//      Parsing:
//      0:0:03. ('h\:m\:ss\.F') --> 00:00:03
//      0:0:03.1 ('h\:m\:ss\.F') --> 00:00:03.100000
//      Cannot parse 0:0:03.12 with 'h\:m\:ss\.F'.
```

[Back to table](#)

The "FF" custom format specifier

The "FF" custom format specifier outputs the hundredths of a second in a time interval. In a formatting operation, any remaining fractional digits are truncated. If there are any trailing fractional zeros, they aren't included in the result string. In a parsing operation that calls the [TimeSpan.ParseExact](#) or [TimeSpan.TryParseExact](#) method, the presence of the tenths and hundredths of a second digit is optional.

The following example uses the "FF" custom format specifier to display the hundredths of a second in a [TimeSpan](#) value. It also uses this custom format specifier in a parsing operation.

C#

```
Console.WriteLine("Formatting:");
TimeSpan ts1 = TimeSpan.Parse("0:0:3.697");
Console.WriteLine("{0} ('FF') --> {0:FF}", ts1);

TimeSpan ts2 = TimeSpan.Parse("0:0:3.809");
Console.WriteLine("{0} ('ss\\.FF') --> {0:ss\\.FF}", ts2);
Console.WriteLine();

Console.WriteLine("Parsing:");
string[] inputs = { "0:0:03.", "0:0:03.1", "0:0:03.127" };
string fmt = @"h\:m\:ss\.FF";
TimeSpan ts3;

foreach (string input in inputs) {
    if (TimeSpan.TryParseExact(input, fmt, null, out ts3))
        Console.WriteLine("{0} ('{1}') --> {2}", input, fmt, ts3);
    else
        Console.WriteLine("Cannot parse {0} with '{1}'.",
                         input, fmt);
}
// The example displays the following output:
//      Formatting:
//      00:00:03.6970000 ('FF') --> 69
//      00:00:03.8090000 ('ss\.FF') --> 03.8
//
//      Parsing:
//      0:0:03. ('h\:m\:ss\.FF') --> 00:00:03
//      0:0:03.1 ('h\:m\:ss\.FF') --> 00:00:03.1000000
//      Cannot parse 0:0:03.127 with 'h\:m\:ss\.FF'.
```

[Back to table](#)

The "FFF" custom format specifier

The "FFF" custom format specifier (with three "F" characters) outputs the milliseconds in a time interval. In a formatting operation, any remaining fractional digits are truncated. If there are any trailing fractional zeros, they aren't included in the result string. In a parsing operation that calls the [TimeSpan.ParseExact](#) or [TimeSpan.TryParseExact](#) method, the presence of the tenths, hundredths, and thousandths of a second digit is optional.

The following example uses the "FFF" custom format specifier to display the thousandths of a second in a [TimeSpan](#) value. It also uses this custom format specifier in a parsing operation.

C#

```

Console.WriteLine("Formatting:");
TimeSpan ts1 = TimeSpan.Parse("0:0:3.6974");
Console.WriteLine("{0} ('FFF') --> {0:FFF}", ts1);

TimeSpan ts2 = TimeSpan.Parse("0:0:3.8009");
Console.WriteLine("{0} ('ss\\.FFF') --> {0:ss\\.FFF}", ts2);
Console.WriteLine();

Console.WriteLine("Parsing:");
string[] inputs = { "0:0:03.", "0:0:03.12", "0:0:03.1279" };
string fmt = @"h\:m\:ss\.FFF";
TimeSpan ts3;

foreach (string input in inputs) {
    if (TimeSpan.TryParseExact(input, fmt, null, out ts3))
        Console.WriteLine("{0} ('{1}') --> {2}", input, fmt, ts3);
    else
        Console.WriteLine("Cannot parse {0} with '{1}'.",
                          input, fmt);
}
// The example displays the following output:
//      Formatting:
//      00:00:03.6974000 ('FFF') --> 697
//      00:00:03.8009000 ('ss\.FFF') --> 03.8
//
//      Parsing:
//      0:0:03. ('h\:m\:ss\.FFF') --> 00:00:03
//      0:0:03.12 ('h\:m\:ss\.FFF') --> 00:00:03.1200000
//      Cannot parse 0:0:03.1279 with 'h\:m\:ss\.FFF'.

```

[Back to table](#)

The "FFFF" custom format specifier

The "FFFF" custom format specifier (with four "F" characters) outputs the ten-thousandths of a second in a time interval. In a formatting operation, any remaining fractional digits are truncated. If there are any trailing fractional zeros, they aren't included in the result string. In a parsing operation that calls the [TimeSpan.ParseExact](#) or [TimeSpan.TryParseExact](#) method, the presence of the tenths, hundredths, thousandths, and ten-thousandths of a second digit is optional.

The following example uses the "FFFF" custom format specifier to display the ten-thousandths of a second in a [TimeSpan](#) value. It also uses the "FFFF" custom format specifier in a parsing operation.

C#

```

Console.WriteLine("Formatting:");
TimeSpan ts1 = TimeSpan.Parse("0:0:3.69749");
Console.WriteLine("{0} ('FFFF') --> {0:FFFF}", ts1);

TimeSpan ts2 = TimeSpan.Parse("0:0:3.80009");
Console.WriteLine("{0} ('ss\\.FFFF') --> {0:ss\\.FFFF}", ts2);
Console.WriteLine();

Console.WriteLine("Parsing:");
string[] inputs = { "0:0:03.", "0:0:03.12", "0:0:03.12795" };
string fmt = @"h\:m\:ss\.FFFF";
TimeSpan ts3;

foreach (string input in inputs) {
    if (TimeSpan.TryParseExact(input, fmt, null, out ts3))
        Console.WriteLine("{0} ('{1}') --> {2}", input, fmt, ts3);
    else
        Console.WriteLine("Cannot parse {0} with '{1}'.",
                          input, fmt);
}
// The example displays the following output:
//      Formatting:
//      00:00:03.6974900 ('FFFF') --> 6974
//      00:00:03.8000900 ('ss\.FFFF') --> 03.8
//
//      Parsing:
//      0:0:03. ('h\:m\:ss\.FFFF') --> 00:00:03
//      0:0:03.12 ('h\:m\:ss\.FFFF') --> 00:00:03.1200000
//      Cannot parse 0:0:03.12795 with 'h\:m\:ss\.FFFF'.

```

[Back to table](#)

The "FFFFF" custom format specifier

The "FFFFF" custom format specifier (with five "F" characters) outputs the hundred-thousandths of a second in a time interval. In a formatting operation, any remaining fractional digits are truncated. If there are any trailing fractional zeros, they aren't included in the result string. In a parsing operation that calls the [TimeSpan.ParseExact](#) or [TimeSpan.TryParseExact](#) method, the presence of the tenths, hundredths, thousandths, ten-thousandths, and hundred-thousandths of a second digit is optional.

The following example uses the "FFFFF" custom format specifier to display the hundred-thousandths of a second in a [TimeSpan](#) value. It also uses the "FFFFF" custom format specifier in a parsing operation.

C#

```

Console.WriteLine("Formatting:");
TimeSpan ts1 = TimeSpan.Parse("0:0:3.697497");
Console.WriteLine("{0} ('FFFFF') --> {0:FFFFF}", ts1);

TimeSpan ts2 = TimeSpan.Parse("0:0:3.800009");
Console.WriteLine("{0} ('ss\\.FFFFF') --> {0:ss\\.FFFFF}", ts2);
Console.WriteLine();

Console.WriteLine("Parsing:");
string[] inputs = { "0:0:03.", "0:0:03.12", "0:0:03.127956" };
string fmt = @"h\:m\:ss\.FFFFF";
TimeSpan ts3;

foreach (string input in inputs) {
    if (TimeSpan.TryParseExact(input, fmt, null, out ts3))
        Console.WriteLine("{0} ('{1}') --> {2}", input, fmt, ts3);
    else
        Console.WriteLine("Cannot parse {0} with '{1}'.",
                          input, fmt);
}
// The example displays the following output:
//      Formatting:
//      00:00:03.6974970 ('FFFFF') --> 69749
//      00:00:03.8000090 ('ss\\.FFFFF') --> 03.8
//
//      Parsing:
//      0:0:03. ('h\:m\:ss\.FFFF') --> 00:00:03
//      0:0:03.12 ('h\:m\:ss\.FFFF') --> 00:00:03.1200000
//      Cannot parse 0:0:03.127956 with 'h\:m\:ss\.FFFF'.

```

[Back to table](#)

The "FFFFFF" custom format specifier

The "FFFFFF" custom format specifier (with six "F" characters) outputs the millionths of a second in a time interval. In a formatting operation, any remaining fractional digits are truncated. If there are any trailing fractional zeros, they aren't included in the result string. In a parsing operation that calls the [TimeSpan.ParseExact](#) or [TimeSpan.TryParseExact](#) method, the presence of the tenths, hundredths, thousandths, ten-thousandths, hundred-thousandths, and millionths of a second digit is optional.

The following example uses the "FFFFFF" custom format specifier to display the millionths of a second in a [TimeSpan](#) value. It also uses this custom format specifier in a parsing operation.

C#

```

Console.WriteLine("Formatting:");
TimeSpan ts1 = TimeSpan.Parse("0:0:3.6974974");
Console.WriteLine("{0} ('FFFFFF') --> {0:FFFFFF}", ts1);

TimeSpan ts2 = TimeSpan.Parse("0:0:3.8000009");
Console.WriteLine("{0} ('ss\\.FFFFFF') --> {0:ss\\.FFFFFF}", ts2);
Console.WriteLine();

Console.WriteLine("Parsing:");
string[] inputs = { "0:0:03.", "0:0:03.12", "0:0:03.1279569" };
string fmt = @"h\:m\:ss\.FFFFFF";
TimeSpan ts3;

foreach (string input in inputs) {
    if (TimeSpan.TryParseExact(input, fmt, null, out ts3))
        Console.WriteLine("{0} ('{1}') --> {2}", input, fmt, ts3);
    else
        Console.WriteLine("Cannot parse {0} with '{1}'.",
                          input, fmt);
}
// The example displays the following output:
//      Formatting:
//      00:00:03.6974974 ('FFFFFF') --> 697497
//      00:00:03.8000009 ('ss\\.FFFFFF') --> 03.8
//
//      Parsing:
//      0:0:03. ('h\:m\:ss\.FFFFFF') --> 00:00:03
//      0:0:03.12 ('h\:m\:ss\.FFFFFF') --> 00:00:03.1200000
//      Cannot parse 0:0:03.1279569 with 'h\:m\:ss\.FFFFFF'.

```

[Back to table](#)

The "FFFFFF" custom format specifier

The "FFFFFF" custom format specifier (with seven "F" characters) outputs the ten-millionths of a second (or the fractional number of ticks) in a time interval. If there are any trailing fractional zeros, they aren't included in the result string. In a parsing operation that calls the [TimeSpan.ParseExact](#) or [TimeSpan.TryParseExact](#) method, the presence of the seven fractional digits in the input string is optional.

The following example uses the "FFFFFF" custom format specifier to display the fractional parts of a second in a [TimeSpan](#) value. It also uses this custom format specifier in a parsing operation.

C#

```

Console.WriteLine("Formatting:");
TimeSpan ts1 = TimeSpan.Parse("0:0:3.6974974");

```

```

Console.WriteLine("{0} ('FFFFFFF') --> {0:FFFFFFF}", ts1);

TimeSpan ts2 = TimeSpan.Parse("0:0:3.9500000");
Console.WriteLine("{0} ('ss\\.FFFFFFF') --> {0:ss\\.FFFFFFF}", ts2);
Console.WriteLine();

Console.WriteLine("Parsing:");
string[] inputs = { "0:0:03.", "0:0:03.12", "0:0:03.1279569" };
string fmt = @"h\:m\:ss\.FFFFFFF";
TimeSpan ts3;

foreach (string input in inputs) {
    if (TimeSpan.TryParseExact(input, fmt, null, out ts3))
        Console.WriteLine("{0} ('{1}') --> {2}", input, fmt, ts3);
    else
        Console.WriteLine("Cannot parse {0} with '{1}'.",
                          input, fmt);
}
// The example displays the following output:
//   Formatting:
//   00:00:03.6974974 ('FFFFFFF') --> 6974974
//   00:00:03.9500000 ('ss\\.FFFFFFF') --> 03.95
//
//   Parsing:
//   0:0:03. ('h\:m\:ss\\.FFFFFFF') --> 00:00:03
//   0:0:03.12 ('h\:m\:ss\\.FFFFFFF') --> 00:00:03.1200000
//   0:0:03.1279569 ('h\:m\:ss\\.FFFFFFF') --> 00:00:03.1279569

```

[Back to table](#)

Other characters

Any other unescaped character in a format string, including a white-space character, is interpreted as a custom format specifier. In most cases, the presence of any other unescaped character results in a [FormatException](#).

There are two ways to include a literal character in a format string:

- Enclose it in single quotation marks (the literal string delimiter).
- Precede it with a backslash ("\"), which is interpreted as an escape character. This means that, in C#, the format string must either be @-quoted, or the literal character must be preceded by an additional backslash.

In some cases, you may have to use conditional logic to include an escaped literal in a format string. The following example uses conditional logic to include a sign symbol for negative time intervals.

```

using System;

public class Example
{
    public static void Main()
    {
        TimeSpan result = new DateTime(2010, 01, 01) - DateTime.Now;
        String fmt = (result < TimeSpan.Zero ? "\\" : "") +
"dd\\\\.hh\\:\\mm";

        Console.WriteLine(result.ToString(fmt));
        Console.WriteLine("Interval: {0:" + fmt + "}", result);
    }
}

// The example displays output like the following:
//      -1291.10:54
//      Interval: -1291.10:54

```

.NET doesn't define a grammar for separators in time intervals. This means that the separators between days and hours, hours and minutes, minutes and seconds, and seconds and fractions of a second must all be treated as character literals in a format string.

The following example uses both the escape character and the single quote to define a custom format string that includes the word "minutes" in the output string.

C#

```

TimeSpan interval = new TimeSpan(0, 32, 45);
// Escape literal characters in a format string.
String fmt = @"mm\:ss\ \m\i\n\u\t\e\s";
Console.WriteLine(interval.ToString(fmt));
// Delimit literal characters in a format string with the ' symbol.
fmt = "mm':'ss' minutes'";
Console.WriteLine(interval.ToString(fmt));
// The example displays the following output:
//      32:45 minutes
//      32:45 minutes

```

[Back to table](#)

See also

- [Formatting Types](#)
- [Standard TimeSpan Format Strings](#)

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

Enumeration format strings

Article • 06/20/2023

You can use the [Enum.ToString](#) method to create a new string object that represents the numeric, hexadecimal, or string value of an enumeration member. This method takes one of the enumeration formatting strings to specify the value that you want returned.

The following sections list the enumeration formatting strings and the values they return. These format specifiers aren't case-sensitive.

G or g

Displays the enumeration entry as a string value, if possible, and otherwise displays the integer value of the current instance. If the enumeration is defined with the [FlagsAttribute](#) set, the string values of each valid entry are concatenated together, separated by commas. If the [Flags](#) attribute isn't set, an invalid value is displayed as a numeric entry. The following example illustrates the `G` format specifier.

C#

```
Console.WriteLine(((DayOfWeek)7).ToString("G"));      // 7
Console.WriteLine(ConsoleColor.Red.ToString("G"));    // Red

var attributes = FileAttributes.Hidden | FileAttributes.Archive;
Console.WriteLine(attributes.ToString("G"));          // Hidden, Archive
```

F or f

Displays the enumeration entry as a string value, if possible. If the value can be displayed as a summation of the entries in the enumeration (even if the [Flags](#) attribute isn't present), the string values of each valid entry are concatenated together, separated by commas. If the value can't be determined by the enumeration entries, then the value is formatted as the integer value. The following example illustrates the `F` format specifier.

C#

```
Console.WriteLine(((DayOfWeek)7).ToString("F"));      // Monday, Saturday
Console.WriteLine(ConsoleColor.Blue.ToString("F"));   // Blue
```

```
var attributes = FileAttributes.Hidden | FileAttributes.Archive;
Console.WriteLine(attributes.ToString("F")); // Hidden, Archive
```

D or d

Displays the enumeration entry as an integer value in the shortest representation possible. The following example illustrates the `D` format specifier.

C#

```
Console.WriteLine(((DayOfWeek)7).ToString("D")); // 7
Console.WriteLine(ConsoleColor.Cyan.ToString("D")); // 11

var attributes = FileAttributes.Hidden | FileAttributes.Archive;
Console.WriteLine(attributes.ToString("D")); // 34
```

X or x

Displays the enumeration entry as a hexadecimal value. The value is represented with leading zeros as necessary, to ensure that the result string has two characters for each byte in the enumeration type's [underlying numeric type](#). The following example illustrates the X format specifier. In the example, the underlying types of `DayOfWeek`, `ConsoleColor` and `FileAttributes` is `Int32`, or a 32-bit (or 4-byte) integer, which produces an 8-character result string.

C#

```
Console.WriteLine(((DayOfWeek)7).ToString("X")); // 00000007
Console.WriteLine(ConsoleColor.Cyan.ToString("X")); // 0000000B

var attributes = FileAttributes.Hidden | FileAttributes.Archive;
Console.WriteLine(attributes.ToString("X")); // 00000022
```

Example

The following example defines an enumeration called `Colors` that consists of three entries: `Red`, `Blue`, and `Green`.

C#

```
public enum Color { Red = 1, Blue = 2, Green = 3 };
```

After the enumeration is defined, an instance can be declared in the following manner.

C#

```
Color myColor = Color.Green;
```

The `Color.ToString(System.String)` method can then be used to display the enumeration value in different ways, depending on the format specifier passed to it.

C#

```
Console.WriteLine("The value of myColor is {0}.",
                  myColor.ToString("G"));
Console.WriteLine("The value of myColor is {0}.",
                  myColor.ToString("F"));
Console.WriteLine("The value of myColor is {0}.",
                  myColor.ToString("D"));
Console.WriteLine("The value of myColor is 0x{0}.",
                  myColor.ToString("X"));
// The example displays the following output to the console:
//      The value of myColor is Green.
//      The value of myColor is Green.
//      The value of myColor is 3.
//      The value of myColor is 0x00000003.
```

See also

- [Formatting types](#)



Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



.NET feedback

The .NET documentation is open source. Provide feedback here.



[Open a documentation issue](#)



[Provide product feedback](#)

Composite formatting

Article • 08/14/2023

The .NET composite formatting feature takes a list of objects and a composite format string as input. A composite format string consists of fixed text intermixed with indexed placeholders, called format items. These format items correspond to the objects in the list. The formatting operation yields a result string that consists of the original fixed text intermixed with the string representation of the objects in the list.

ⓘ Important

Instead of using composite format strings, you can use *interpolated strings* if the language and its version that you're using support them. An interpolated string contains *interpolated expressions*. Each interpolated expression is resolved with the expression's value and included in the result string when the string is assigned. For more information, see [String interpolation \(C# Reference\)](#) and [Interpolated strings \(Visual Basic Reference\)](#).

The following methods support the composite formatting feature:

- [String.Format](#), which returns a formatted result string.
- [StringBuilder.AppendFormat](#), which appends a formatted result string to a [StringBuilder](#) object.
- Some overloads of the [Console.WriteLine](#) method, which display a formatted result string to the console.
- Some overloads of the [TextWriter.WriteLine](#) method, which write the formatted result string to a stream or file. The classes derived from [TextWriter](#), such as [StreamWriter](#) and [HtmlTextWriter](#), also share this functionality.
- [Debug.WriteLine\(String, Object\[\]\)](#), which outputs a formatted message to trace listeners.
- The [Trace.TraceError\(String, Object\[\]\)](#), [Trace.TraceInformation\(String, Object\[\]\)](#), and [Trace.TraceWarning\(String, Object\[\]\)](#) methods, which output formatted messages to trace listeners.
- The [TraceSource.TraceInformation\(String, Object\[\]\)](#) method, which writes an informational method to trace listeners.

Composite format string

A composite format string and object list are used as arguments of methods that support the composite formatting feature. A composite format string consists of zero or more runs of fixed text intermixed with one or more format items. The fixed text is any string that you choose, and each format item corresponds to an object or boxed structure in the list. The string representation of each object replaces the corresponding format item.

Consider the following [Format](#) code fragment:

C#

```
string.Format("Name = {0}, hours = {1:hh}", "Fred", DateTime.Now);
```

The fixed text is `Name =` and `, hours =`. The format items are `{0}`, whose index of 0 corresponds to the object `name`, and `{1:hh}`, whose index of 1 corresponds to the object `DateTime.Now`.

Format item syntax

Each format item takes the following form and consists of the following components:

```
{index[,alignment][:formatString]}
```

The matching braces (`{` and `}`) are required.

Index component

The mandatory *index* component, which is also called a parameter specifier, is a number starting from 0 that identifies a corresponding item in the list of objects. That is, the format item whose parameter specifier is `0` formats the first object in the list. The format item whose parameter specifier is `1` formats the second object in the list, and so on. The following example includes four parameter specifiers, numbered zero through three, to represent prime numbers less than 10:

C#

```
string.Format("Name = {0}, hours = {1:hh}", "Fred", DateTime.Now);
```

Multiple format items can refer to the same element in the list of objects by specifying the same parameter specifier. For example, you can format the same numeric value in

hexadecimal, scientific, and number format by specifying a composite format string such as `"0x{0:X} {0:E} {0:N}"`, as the following example shows:

```
C#
```

```
string multiple = string.Format("0x{0:X} {0:E} {0:N}",
                                 Int64.MaxValue);
Console.WriteLine(multiple);

// The example displays the following output:
//      0x7FFFFFFFFFFFFF 9.223372E+018 9,223,372,036,854,775,807.00
```

Each format item can refer to any object in the list. For example, if there are three objects, you can format the second, first, and third object by specifying a composite format string such as `{1} {0} {2}`. An object that isn't referenced by a format item is ignored. A [FormatException](#) is thrown at run time if a parameter specifier designates an item outside the bounds of the list of objects.

Alignment component

The optional *alignment* component is a signed integer indicating the preferred formatted field width. If the value of *alignment* is less than the length of the formatted string, *alignment* is ignored, and the length of the formatted string is used as the field width. The formatted data in the field is right-aligned if *alignment* is positive and left-aligned if *alignment* is negative. If padding is necessary, white space is used. The comma is required if *alignment* is specified.

The following example defines two arrays, one containing the names of employees and the other containing the hours they worked over two weeks. The composite format string left-aligns the names in a 20-character field and right-aligns their hours in a 5-character field. The "N1" standard format string formats the hours with one fractional digit.

```
C#
```

```
string[] names = { "Adam", "Bridgette", "Carla", "Daniel",
                   "Ebenezer", "Francine", "George" };
decimal[] hours = { 40, 6.667m, 40.39m, 82,
                     40.333m, 80, 16.75m };

Console.WriteLine("{0,-20} {1,5}\n", "Name", "Hours");

for (int counter = 0; counter < names.Length; counter++)
    Console.WriteLine("{0,-20} {1,5:N1}", names[counter], hours[counter]);

// The example displays the following output:
```

```
//      Name          Hours
//      Adam          40.0
//      Bridgette    6.7
//      Carla         40.4
//      Daniel        82.0
//      Ebenezer     40.3
//      Francine      80.0
//      George        16.8
```

Format string component

The optional *formatString* component is a format string that's appropriate for the type of object being formatted. You can specify:

- A standard or custom numeric format string if the corresponding object is a numeric value.
- A standard or custom date and time format string if the corresponding object is a [DateTime](#) object.
- An [enumeration format string](#) if the corresponding object is an enumeration value.

If *formatString* isn't specified, the general ("G") format specifier for a numeric, date and time, or enumeration type is used. The colon is required if *formatString* is specified.

The following table lists types or categories of types in the .NET class library that support a predefined set of format strings, and provides links to the articles that list the supported format strings. String formatting is an extensible mechanism that makes it possible to define new format strings for all existing types and to define a set of format strings supported by an application-defined type.

For more information, see the [IFormattable](#) and [ICustomFormatter](#) interface articles.

| Type or type category | See |
|--|---|
| Date and time types (DateTime , DateTimeOffset) | Standard Date and Time Format Strings |
| | Custom Date and Time Format Strings |
| Enumeration types (all types derived from System.Enum) | Enumeration Format Strings |
| Numeric types (BigInteger , Byte , Decimal , Double , Int16 , Int32 , Int64 , SByte , Single , UInt16 , UInt32 , UInt64) | Standard Numeric Format Strings |

| Type or type category | See |
|-----------------------|----------------------------------|
| | Custom Numeric Format Strings |
| Guid | Guid.ToString(String) |
| TimeSpan | Standard TimeSpan Format Strings |
| | Custom TimeSpan Format Strings |

Escaping braces

Opening and closing braces are interpreted as starting and ending a format item. To display a literal opening brace or closing brace, you must use an escape sequence. Specify two opening braces ({{}) in the fixed text to display one opening brace ({}), or two closing braces (}}) to display one closing brace (}).

Escaped braces with a format item are parsed differently between .NET and .NET Framework.

.NET

Braces can be escaped around a format item. For example, consider the format item {{{0:D}}} , which is intended to display an opening brace, a numeric value formatted as a decimal number, and a closing brace. The format item is interpreted in the following manner:

1. The first two opening braces ({{}) are escaped and yield one opening brace.
2. The next three characters ({{0:}) are interpreted as the start of a format item.
3. The next character (D) is interpreted as the Decimal standard numeric format specifier.
4. The next brace (}) is interpreted as the end of the format item.
5. The final two closing braces are escaped and yield one closing brace.
6. The final result that's displayed is the literal string, {{6324}}.

C#

```
int value = 6324;
string output = string.Format("{{{{0:D}}}}", value);

Console.WriteLine(output);
```

```
// The example displays the following output:  
// {6324}
```

.NET Framework

Braces in a format item are interpreted sequentially in the order they're encountered. Interpreting nested braces isn't supported.

The way escaped braces are interpreted can lead to unexpected results. For example, consider the format item `{{{0:D}}}`, which is intended to display an opening brace, a numeric value formatted as a decimal number, and a closing brace. However, the format item is interpreted in the following manner:

1. The first two opening braces (`{{`) are escaped and yield one opening brace.
2. The next three characters (`{0: D}`) are interpreted as the start of a format item.
3. The next character (`D`) would be interpreted as the Decimal standard numeric format specifier, but the next two escaped braces (`}}`) yield a single brace. Because the resulting string (`D}`) isn't a standard numeric format specifier, the resulting string is interpreted as a custom format string that means display the literal string `D}` .
4. The last brace (`}`) is interpreted as the end of the format item.
5. The final result that's displayed is the literal string, `{D}`. The numeric value that was to be formatted isn't displayed.

C#

```
int value = 6324;  
string output = string.Format("{{{0:D}}}",  
                             value);  
Console.WriteLine(output);  
  
// The example displays the following output:  
// {D}
```

One way to write your code to avoid misinterpreting escaped braces and format items is to format the braces and format items separately. That is, in the first format operation, display a literal opening brace. In the next operation, display the result of the format item, and in the final operation, display a literal closing brace. The following example illustrates this approach:

C#

```
int value = 6324;
string output = string.Format("{0}{1:D}{2}",
                             "{", value, "}");
Console.WriteLine(output);

// The example displays the following output:
//      {6324}
```

Processing order

If the call to the composite formatting method includes an [IFormatProvider](#) argument whose value isn't `null`, the runtime calls its [IFormatProvider.GetFormat](#) method to request an [ICustomFormatter](#) implementation. If the method can return an [ICustomFormatter](#) implementation, it's cached during the call of the composite formatting method.

Each value in the parameter list that corresponds to a format item is converted to a string as follows:

1. If the value to be formatted is `null`, an empty string [String.Empty](#) is returned.
2. If an [ICustomFormatter](#) implementation is available, the runtime calls its [Format](#) method. The runtime passes the format item's `formatString` value (or `null` if it's not present) to the method. The runtime also passes the [IFormatProvider](#) implementation to the method. If the call to the [ICustomFormatter.Format](#) method returns `null`, execution proceeds to the next step. Otherwise, the result of the [ICustomFormatter.Format](#) call is returned.
3. If the value implements the [IFormattable](#) interface, the interface's [ToString\(String, IFormatProvider\)](#) method is called. If one is present in the format item, the `formatString` value is passed to the method. Otherwise, `null` is passed. The [IFormatProvider](#) argument is determined as follows:
 - For a numeric value, if a composite formatting method with a non-null [IFormatProvider](#) argument is called, the runtime requests a [NumberFormatInfo](#) object from its [IFormatProvider.GetFormat](#) method. If it's unable to supply one, if the value of the argument is `null`, or if the composite formatting method doesn't have an [IFormatProvider](#) parameter, the [NumberFormatInfo](#) object for the current culture is used.
 - For a date and time value, if a composite formatting method with a non-null [IFormatProvider](#) argument is called, the runtime requests a [DateTimeFormatInfo](#) object from its [IFormatProvider.GetFormat](#) method. In

the following situations, the [DateTimeFormatInfo](#) object for the current culture is used instead:

- The [IFormatProvider.GetFormat](#) method is unable to supply a [DateTimeFormatInfo](#) object.
 - The value of the argument is `null`.
 - The composite formatting method doesn't have an [IFormatProvider](#) parameter.
- For objects of other types, if a composite formatting method is called with an [IFormatProvider](#) argument, its value is passed directly to the [IFormattable.ToString](#) implementation. Otherwise, `null` is passed to the [IFormattable.ToString](#) implementation.

4. The type's parameterless `ToString` method, which either overrides [Object.ToString\(\)](#) or inherits the behavior of its base class, is called. In this case, the format string specified by the `formatString` component in the format item, if it's present, is ignored.

Alignment is applied after the preceding steps have been performed.

Code examples

The following example shows one string created using composite formatting and another created using an object's `ToString` method. Both types of formatting produce equivalent results.

C#

```
string formatString1 = string.Format("{0:ddd MMMM}", DateTime.Now);
string formatString2 = DateTime.Now.ToString("ddd MMMM");
```

Assuming that the current day is a Thursday in May, the value of both strings in the preceding example is `Thursday May` in the U.S. English culture.

[Console.WriteLine](#) exposes the same functionality as [String.Format](#). The only difference between the two methods is that [String.Format](#) returns its result as a string, while [Console.WriteLine](#) writes the result to the output stream associated with the [Console](#) object. The following example uses the [Console.WriteLine](#) method to format the value of `myNumber` to a currency value:

C#

```
int myNumber = 100;
Console.WriteLine("{0:C}", myNumber);

// The example displays the following output
// if en-US is the current culture:
//      $100.00
```

The following example demonstrates formatting multiple objects, including formatting one object in two different ways:

C#

```
string myName = "Fred";
Console.WriteLine(string.Format("Name = {0}, hours = {1:hh}, minutes =
{1:mm}",
                               myName, DateTime.Now));

// Depending on the current time, the example displays output like the
// following:
//      Name = Fred, hours = 11, minutes = 30
```

The following example demonstrates the use of alignment in formatting. The arguments that are formatted are placed between vertical bar characters (|) to highlight the resulting alignment.

C#

```
string firstName = "Fred";
string lastName = "Opals";
int myNumber = 100;

string formatFirstName = string.Format("First Name = |{0,10}|", firstName);
string formatLastName = string.Format("Last Name = |{0,10}|", lastName);
string formatPrice = string.Format("Price = |{0,10:C}|", myNumber);
Console.WriteLine(formatFirstName);
Console.WriteLine(formatLastName);
Console.WriteLine(formatPrice);
Console.WriteLine();

formatFirstName = string.Format("First Name = |{0,-10}|", firstName);
formatLastName = string.Format("Last Name = |{0,-10}|", lastName);
formatPrice = string.Format("Price = |{0,-10:C}|", myNumber);
Console.WriteLine(formatFirstName);
Console.WriteLine(formatLastName);
Console.WriteLine(formatPrice);

// The example displays the following output on a system whose current
// culture is en-US:
//      First Name = |      Fred|
//      Last Name = |      Opals|
```

```
//      Price =      | $100.00|
//  
//      First Name = |Fred      |
//      Last Name =  |Opals     |
//      Price =       |$100.00|
```

See also

- [WriteLine](#)
- [String.Format](#)
- [String interpolation \(C#\)](#)
- [String interpolation \(Visual Basic\)](#)
- [Formatting Types](#)
- [Standard Numeric Format Strings](#)
- [Custom Numeric Format Strings](#)
- [Standard Date and Time Format Strings](#)
- [Custom Date and Time Format Strings](#)
- [Standard TimeSpan Format Strings](#)
- [Custom TimeSpan Format Strings](#)
- [Enumeration Format Strings](#)

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

How to: Pad a Number with Leading Zeros

Article • 09/08/2022

You can add leading zeros to an integer by using the "D" [standard numeric format string](#) with a precision specifier. You can add leading zeros to both integer and floating-point numbers by using a [custom numeric format string](#). This article shows how to use both methods to pad a number with leading zeros.

To pad an integer with leading zeros to a specific length

1. Determine the minimum number of digits you want the integer value to display. Include any leading digits in this number.
2. Determine whether you want to display the integer as a decimal or hexadecimal value.
 - To display the integer as a decimal value, call its `ToString(String)` method, and pass the string "Dn" as the value of the `format` parameter, where *n* represents the minimum length of the string.
 - To display the integer as a hexadecimal value, call its `ToString(String)` method and pass the string "Xn" as the value of the `format` parameter, where *n* represents the minimum length of the string.

You can also use the format string in an interpolated string in both [C#](#) and [Visual Basic](#). Alternatively, you can call a method such as `String.Format` or `Console.WriteLine` that uses [composite formatting](#).

The following example formats several integer values with leading zeros so that the total length of the formatted number is at least eight characters.

C#

```
byte byteValue = 254;
short shortValue = 10342;
int intValue = 1023983;
long lngValue = 6985321;
ulong ulngValue = UInt64.MaxValue;

// Display integer values by calling the ToString method.
```

```

Console.WriteLine("{0,22} {1,22}", byteValue.ToString("D8"),
byteValue.ToString("X8"));
Console.WriteLine("{0,22} {1,22}", shortValue.ToString("D8"),
shortValue.ToString("X8"));
Console.WriteLine("{0,22} {1,22}", intValue.ToString("D8"),
intValue.ToString("X8"));
Console.WriteLine("{0,22} {1,22}", lngValue.ToString("D8"),
lngValue.ToString("X8"));
Console.WriteLine("{0,22} {1,22}", ulngValue.ToString("D8"),
ulngValue.ToString("X8"));
Console.WriteLine();

// Display the same integer values by using composite formatting.
Console.WriteLine("{0,22:D8} {0,22:X8}", byteValue);
Console.WriteLine("{0,22:D8} {0,22:X8}", shortValue);
Console.WriteLine("{0,22:D8} {0,22:X8}", intValue);
Console.WriteLine("{0,22:D8} {0,22:X8}", lngValue);
Console.WriteLine("{0,22:D8} {0,22:X8}", ulngValue);
// The example displays the following output:
//          00000254          000000FE
//          00010342          00002866
//          01023983          000F9FEF
//          06985321          006A9669
//      18446744073709551615      FFFFFFFFFFFFFF
//
//          00000254          000000FE
//          00010342          00002866
//          01023983          000F9FEF
//          06985321          006A9669
//      18446744073709551615      FFFFFFFFFFFFFF
//      18446744073709551615      FFFFFFFFFFFFFF

```

To pad an integer with a specific number of leading zeros

1. Determine how many leading zeros you want the integer value to display.
2. Determine whether you want to display the integer as a decimal or a hexadecimal value.
 - Formatting it as a decimal value requires the "D" standard format specifier.
 - Formatting it as a hexadecimal value requires the "X" standard format specifier.
3. Determine the length of the unpadded numeric string by calling the integer value's `ToString("D").Length` or `ToString("X").Length` method.

4. Add to the length of the unpadded numeric string the number of leading zeros that you want in the formatted string. The result is the total length of the padded string.
5. Call the integer value's `ToString(String)` method, and pass the string "Dn" for decimal strings and "Xn" for hexadecimal strings, where *n* represents the total length of the padded string. You can also use the "Dn" or "Xn" format string in a method that supports composite formatting.

The following example pads an integer value with five leading zeros:

C#

```
int value = 160934;
int decimalLength = value.ToString("D").Length + 5;
int hexLength = value.ToString("X").Length + 5;
Console.WriteLine(value.ToString("D" + decimalLength.ToString()));
Console.WriteLine(value.ToString("X" + hexLength.ToString()));
// The example displays the following output:
//      00000160934
//      00000274A6
```

To pad a numeric value with leading zeros to a specific length

1. Determine how many digits to the left of the decimal you want the string representation of the number to have. Include any leading zeros in this total number of digits.
2. Define a custom numeric format string that uses the zero placeholder ("0") to represent the minimum number of zeros.
3. Call the number's `ToString(String)` method and pass it the custom format string. You can also use the custom format string with string interpolation or a method that supports composite formatting.

The following example formats several numeric values with leading zeros. As a result, the total length of the formatted number is at least eight digits to the left of the decimal.

C#

```
string fmt = "00000000.##";
int intValue = 1053240;
```

```

decimal decValue = 103932.52m;
float sngValue = 1549230.10873992f;
double dblValue = 9034521202.93217412;

// Display the numbers using the ToString method.
Console.WriteLine(intValue.ToString(fmt));
Console.WriteLine(decValue.ToString(fmt));
Console.WriteLine(sngValue.ToString(fmt));
Console.WriteLine(dblValue.ToString(fmt));
Console.WriteLine();

// Display the numbers using composite formatting.
string formatString = " {0,15:" + fmt + "}";
Console.WriteLine(formatString, intValue);
Console.WriteLine(formatString, decValue);
Console.WriteLine(formatString, sngValue);
Console.WriteLine(formatString, dblValue);
// The example displays the following output:
//      01053240
//      00103932.52
//      01549230
//      9034521202.93
//
//          01053240
//          00103932.52
//          01549230
//          9034521202.93

```

To pad a numeric value with a specific number of leading zeros

1. Determine how many leading zeros you want the numeric value to have.
2. Determine the number of digits to the left of the decimal in the unpadded numeric string:
 - a. Determine whether the string representation of a number includes a decimal point symbol.
 - b. If it does include a decimal point symbol, determine the number of characters to the left of the decimal point. If it doesn't include a decimal point symbol, determine the string's length.
3. Create a custom format string that uses:
 - The zero placeholder ("0") for each of the leading zeros to appear in the string.

- Either the zero placeholder or the digit placeholder "#" to represent each digit in the default string.

4. Supply the custom format string as a parameter either to the number's `ToString(String)` method or to a method that supports composite formatting.

The following example pads two `Double` values with five leading zeros:

C#

```
double[] dblValues = { 9034521202.93217412, 9034521202 };
foreach (double dblValue in dblValues)
{
    string decSeparator =
System.Globalization.NumberFormatInfo.CurrentInfo.NumberDecimalSeparator;
    string fmt, formatString;

    if (dblValue.ToString().Contains(decSeparator))
    {
        int digits = dblValue.ToString().IndexOf(decSeparator);
        fmt = new String('0', 5) + new String('#', digits) + ".##";
    }
    else
    {
        fmt = new String('0', dblValue.ToString().Length);
    }
    formatString = "{0,20:" + fmt + "}";

    Console.WriteLine(dblValue.ToString(fmt));
    Console.WriteLine(formatString, dblValue);
}
// The example displays the following output:
//      000009034521202.93
//      000009034521202.93
//      9034521202
//                  9034521202
```

See also

- [Custom Numeric Format Strings](#)
- [Standard Numeric Format Strings](#)
- [Composite Formatting](#)

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

How to: Extract the Day of the Week from a Specific Date

Article • 10/04/2022

.NET makes it easy to determine the ordinal day of the week for a particular date, and to display the localized weekday name for a particular date. An enumerated value that indicates the day of the week corresponding to a particular date is available from the [DayOfWeek](#) or [DayOfWeek](#) property. In contrast, retrieving the weekday name is a formatting operation that can be performed by calling a formatting method, such as a date and time value's `ToString` method or the `String.Format` method. This article shows how to perform these formatting operations.

Extract a number indicating the day of the week

1. Use the static `DateTime.Parse` or `DateTimeOffset.Parse` method to convert the string representation of a date to a `DateTime` or a `DateTimeOffset` value.
2. Use the `DateTime.DayOfWeek` or `DateTimeOffset.DayOfWeek` property to retrieve a `DayOfWeek` value that indicates the day of the week.
3. If necessary, cast (in C#) or convert (in Visual Basic) the `DayOfWeek` value to an integer.

The following example displays an integer that represents the day of the week of a specific date:

C#

```
using System;

public class Example
{
    public static void Main()
    {
        DateTime dateValue = new DateTime(2008, 6, 11);
        Console.WriteLine((int) dateValue.DayOfWeek);
    }
} // The example displays the following output:
//      3
```

Extract the abbreviated weekday name

1. Use the static [DateTime.Parse](#) or [DateTimeOffset.Parse](#) method to convert the string representation of a date to a [DateTime](#) or a [DateTimeOffset](#) value.
2. You can extract the abbreviated weekday name of the current culture or of a specific culture:
 - a. To extract the abbreviated weekday name for the current culture, call the date and time value's [DateTime.ToString\(String\)](#) or [DateTimeOffset.ToString\(String\)](#) instance method, and pass the string `ddd` as the `format` parameter. The following example illustrates the call to the [ToString\(String\)](#) method:

C#

```
using System;

public class Example
{
    public static void Main()
    {
        DateTime dateValue = new DateTime(2008, 6, 11);
        Console.WriteLine(dateValue.ToString("ddd"));
    }
}
// The example displays the following output:
//      Wed
```

- a. To extract the abbreviated weekday name for a specific culture, call the date and time value's [DateTime.ToString\(String, IFormatProvider\)](#) or [DateTimeOffset.ToString\(String, IFormatProvider\)](#) instance method. Pass the string `ddd` as the `format` parameter. Pass either a [CultureInfo](#) or a [DateTimeFormatInfo](#) object that represents the culture whose weekday name you want to retrieve as the `provider` parameter. The following code illustrates a call to the [ToString\(String, IFormatProvider\)](#) method using a [CultureInfo](#) object that represents the fr-FR culture:

C#

```
using System;
using System.Globalization;

public class Example
{
    public static void Main()
    {
        DateTime dateValue = new DateTime(2008, 6, 11);
```

```

        Console.WriteLine(dateValue.ToString("ddd",
            new CultureInfo("fr-FR")));
    }
}
// The example displays the following output:
//      mer.

```

Extract the full weekday name

1. Use the static [DateTime.Parse](#) or [DateTimeOffset.Parse](#) method to convert the string representation of a date to a [DateTime](#) or a [DateTimeOffset](#) value.
2. You can extract the full weekday name of the current culture or of a specific culture:
 - a. To extract the weekday name for the current culture, call the date and time value's [DateTime.ToString\(String\)](#) or [DateTimeOffset.ToString\(String\)](#) instance method, and pass the string `dddd` as the `format` parameter. The following example illustrates the call to the [ToString\(String\)](#) method:

C#

```

using System;

public class Example
{
    public static void Main()
    {
        DateTime dateValue = new DateTime(2008, 6, 11);
        Console.WriteLine(dateValue.ToString("dddd"));
    }
}
// The example displays the following output:
//      Wednesday

```

- b. To extract the weekday name for a specific culture, call the date and time value's [DateTime.ToString\(String, IFormatProvider\)](#) or [DateTimeOffset.ToString\(String, IFormatProvider\)](#) instance method. Pass the string `dddd` as the `format` parameter. Pass either a [CultureInfo](#) or a [DateTimeFormatInfo](#) object that represents the culture whose weekday name you want to retrieve as the `provider` parameter. The following code illustrates a call to the [ToString\(String, IFormatProvider\)](#) method using a [CultureInfo](#) object that represents the es-ES culture:

C#

```

using System;
using System.Globalization;

public class Example
{
    public static void Main()
    {
        DateTime dateValue = new DateTime(2008, 6, 11);
        Console.WriteLine(dateValue.ToString("dddd",
                                             new CultureInfo("es-ES")));
    }
}
// The example displays the following output:
//       miércoles.

```

Example

The following example illustrates calls to the [DateTime.DayOfWeek](#) and [DateTimeOffset.DayOfWeek](#) properties to retrieve the number that represents the day of the week for a particular date. It also includes calls to the [DateTime.ToString](#) and [DateTimeOffset.ToString](#) methods to extract the abbreviated weekday name and the full weekday name.

C#

```

using System;
using System.Globalization;

public class Example
{
    public static void Main()
    {
        string dateString = "6/11/2007";
        DateTime dateValue;
        DateTimeOffset dateOffsetValue;

        try
        {
            DateTimeFormatInfo dateTimeFormats;
            // Convert date representation to a date value
            dateValue = DateTime.Parse(dateString,
                                       CultureInfo.InvariantCulture);
            dateOffsetValue = new DateTimeOffset(dateValue,
                                                 TimeZoneInfo.Local.GetUtcOffset(dateValue));

            // Convert date representation to a number indicating the day of
            // week
            Console.WriteLine((int) dateValue.DayOfWeek);
        }
    }
}

```

```

        Console.WriteLine((int) dateOffsetValue.DayOfWeek);

        // Display abbreviated weekday name using current culture
        Console.WriteLine(dateValue.ToString("ddd"));
        Console.WriteLine(dateOffsetValue.ToString("ddd"));

        // Display full weekday name using current culture
        Console.WriteLine(dateValue.ToString("ddd"));
        Console.WriteLine(dateOffsetValue.ToString("ddd"));

        // Display abbreviated weekday name for de-DE culture
        Console.WriteLine(dateValue.ToString("ddd", new CultureInfo("de-
DE")));
        Console.WriteLine(dateOffsetValue.ToString("ddd",
                                         new CultureInfo("de-
DE")));

        // Display abbreviated weekday name with de-DE DateTimeFormatInfo
object
        dateTimeFormats = new CultureInfo("de-DE").DateTimeFormat;
        Console.WriteLine(dateValue.ToString("ddd", dateTimeFormats));
        Console.WriteLine(dateOffsetValue.ToString("ddd",
                                         dateTimeFormats));

        // Display full weekday name for fr-FR culture
        Console.WriteLine(dateValue.ToString("ddd", new CultureInfo("fr-
FR")));
        Console.WriteLine(dateOffsetValue.ToString("ddd",
                                         new CultureInfo("fr-
FR")));

        // Display abbreviated weekday name with fr-FR DateTimeFormatInfo
object
        dateTimeFormats = new CultureInfo("fr-FR").DateTimeFormat;
        Console.WriteLine(dateValue.ToString("ddd", dateTimeFormats));
        Console.WriteLine(dateOffsetValue.ToString("ddd",
                                         dateTimeFormats));
    }
    catch (FormatException)
    {
        Console.WriteLine("Unable to convert {0} to a date.", dateString);
    }
}
// The example displays the following output:
//      1
//      1
//      Mon
//      Mon
//      Monday
//      Monday
//      Mo
//      Mo
//      Mo
//      Mo

```

```
//      lun.  
//      lun.  
//      lundi  
//      lundi
```

Individual languages might provide functionality that duplicates or supplements the functionality provided by .NET. For example, Visual Basic includes two such functions:

- `Weekday`, which returns a number that indicates the day of the week of a particular date. It considers the ordinal value of the first day of the week to be one, whereas the `DateTime.DayOfWeek` property considers it to be zero.
- `WeekdayName`, which returns the name of the week in the current culture that corresponds to a particular weekday number.

The following example illustrates the use of the Visual Basic `Weekday` and `WeekdayName` functions:

VB

```
Imports System.Globalization  
Imports System.Threading  
  
Module Example  
    Public Sub Main()  
        Dim dateValue As Date = #6/11/2008#  
  
        ' Get weekday number using Visual Basic Weekday function  
        Console.WriteLine(Weekday(dateValue))                      ' Displays 4  
        ' Compare with .NET DateTime.DayOfWeek property  
        Console.WriteLine(dateValue.DayOfWeek)                     ' Displays 3  
  
        ' Get weekday name using Weekday and WeekdayName functions  
        Console.WriteLine(WeekdayName(Weekday(dateValue)))       ' Displays  
Wednesday  
  
        ' Change culture to de-DE  
        Dim originalCulture As CultureInfo =  
Thread.CurrentCulture  
        Thread.CurrentCulture = New CultureInfo("de-DE")  
        ' Get weekday name using Weekday and WeekdayName functions  
        Console.WriteLine(WeekdayName(Weekday(dateValue)))       ' Displays  
Donnerstag  
  
        ' Restore original culture  
        Thread.CurrentCulture = originalCulture  
    End Sub  
End Module
```

You can also use the value returned by the `DateTime.DayOfWeek` property to retrieve the weekday name of a particular date. This process requires only a call to the `ToString` method on the `DayOfWeek` value returned by the property. However, this technique doesn't produce a localized weekday name for the current culture, as the following example illustrates:

C#

```
using System;
using System.Globalization;
using System.Threading;

public class Example
{
    public static void Main()
    {
        // Change current culture to fr-FR
        CultureInfo originalCulture = Thread.CurrentThread.CurrentCulture;
        Thread.CurrentThread.CurrentCulture = new CultureInfo("fr-FR");

        DateTime dateValue = new DateTime(2008, 6, 11);
        // Display the DayOfWeek string representation
        Console.WriteLine(dateValue.DayOfWeek.ToString());
        // Restore original current culture
        Thread.CurrentThread.CurrentCulture = originalCulture;
    }
}
// The example displays the following output:
//       Wednesday
```

See also

- [Standard Date and Time Format Strings](#)
- [Custom Date and Time Format Strings](#)

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

How to: Define and Use Custom Numeric Format Providers

Article • 09/15/2021

.NET gives you extensive control over the string representation of numeric values. It supports the following features for customizing the format of numeric values:

- Standard numeric format strings, which provide a predefined set of formats for converting numbers to their string representation. You can use them with any numeric formatting method, such as `Decimal.ToString(String)`, that has a `format` parameter. For details, see [Standard Numeric Format Strings](#).
- Custom numeric format strings, which provide a set of symbols that can be combined to define custom numeric format specifiers. They can also be used with any numeric formatting method, such as `Decimal.ToString(String)`, that has a `format` parameter. For details, see [Custom Numeric Format Strings](#).
- Custom `CultureInfo` or `NumberFormatInfo` objects, which define the symbols and format patterns used in displaying the string representations of numeric values. You can use them with any numeric formatting method, such as `ToString`, that has a `provider` parameter. Typically, the `provider` parameter is used to specify culture-specific formatting.

In some cases (such as when an application must display a formatted account number, an identification number, or a postal code) these three techniques are inappropriate. .NET also enables you to define a formatting object that is neither a `CultureInfo` nor a `NumberFormatInfo` object to determine how a numeric value is formatted. This topic provides the step-by-step instructions for implementing such an object, and provides an example that formats telephone numbers.

Define a custom format provider

1. Define a class that implements the `IFormatProvider` and `ICustomFormatter` interfaces.
2. Implement the `IFormatProvider.GetFormat` method. `GetFormat` is a callback method that the formatting method (such as the `String.Format(IFormatProvider, String, Object[])` method) invokes to retrieve the object that is actually responsible for performing custom formatting. A typical implementation of `GetFormat` does the following:

- a. Determines whether the `Type` object passed as a method parameter represents an `ICustomFormatter` interface.
 - b. If the parameter does represent the `ICustomFormatter` interface, `GetFormat` returns an object that implements the `ICustomFormatter` interface that is responsible for providing custom formatting. Typically, the custom formatting object returns itself.
 - c. If the parameter does not represent the `ICustomFormatter` interface, `GetFormat` returns `null`.
3. Implement the `Format` method. This method is called by the `String.Format(IFormatProvider, String, Object[])` method and is responsible for returning the string representation of a number. Implementing the method typically involves the following:
- a. Optionally, make sure that the method is legitimately intended to provide formatting services by examining the `provider` parameter. For formatting objects that implement both `IFormatProvider` and `ICustomFormatter`, this involves testing the `provider` parameter for equality with the current formatting object.
 - b. Determine whether the formatting object should support custom format specifiers. (For example, an "N" format specifier might indicate that a U.S. telephone number should be output in NANP format, and an "I" might indicate output in ITU-T Recommendation E.123 format.) If format specifiers are used, the method should handle the specific format specifier. It is passed to the method in the `format` parameter. If no specifier is present, the value of the `format` parameter is `String.Empty`.
 - c. Retrieve the numeric value passed to the method as the `arg` parameter. Perform whatever manipulations are required to convert it to its string representation.
 - d. Return the string representation of the `arg` parameter.

Use a custom numeric formatting object

1. Create a new instance of the custom formatting class.
2. Call the `String.Format(IFormatProvider, String, Object[])` formatting method, passing it the custom formatting object, the formatting specifier (or `String.Empty`, if one is not used), and the numeric value to be formatted.

Example

The following example defines a custom numeric format provider named `TelephoneFormatter` that converts a number that represents a U.S. telephone number to its NANP or E.123 format. The method handles two format specifiers, "N" (which outputs the NANP format) and "I" (which outputs the international E.123 format).

C#

```
using System;
using System.Globalization;

public class TelephoneFormatter : IFormatProvider, ICustomFormatter
{
    public object GetFormat(Type formatType)
    {
        if (formatType == typeof(ICustomFormatter))
            return this;
        else
            return null;
    }

    public string Format(string format, object arg, IFormatProvider
formatProvider)
    {
        // Check whether this is an appropriate callback
        if (! this.Equals(formatProvider))
            return null;

        // Set default format specifier
        if (string.IsNullOrEmpty(format))
            format = "N";

        string numericString = arg.ToString();

        if (format == "N")
        {
            if (numericString.Length <= 4)
                return numericString;
            else if (numericString.Length == 7)
                return numericString.Substring(0, 3) + "-" +
numericString.Substring(3, 4);
            else if (numericString.Length == 10)
                return "(" + numericString.Substring(0, 3) + ")" +
numericString.Substring(3, 3) + "-" +
numericString.Substring(6);
            else
                throw new FormatException(
                    string.Format("{0}' cannot be used to format {1}.",
format, arg.ToString()));
        }
        else if (format == "I")
    }
```

```

    {
        if (numericString.Length < 10)
            throw new FormatException(string.Format("{0} does not have 10
digits.", arg.ToString()));
        else
            numericString = "+1 " + numericString.Substring(0, 3) + " " +
numericString.Substring(3, 3) + " " + numericString.Substring(6);
    }
    else
    {
        throw new FormatException(string.Format("The {0} format specifier
is invalid.", format));
    }
    return numericString;
}
}

public class TestTelephoneFormatter
{
    public static void Main()
    {
        Console.WriteLine(String.Format(new TelephoneFormatter(), "{0}", 0));
        Console.WriteLine(String.Format(new TelephoneFormatter(), "{0}",
911));
        Console.WriteLine(String.Format(new TelephoneFormatter(), "{0}",
8490216));
        Console.WriteLine(String.Format(new TelephoneFormatter(), "{0}",
4257884748));

        Console.WriteLine(String.Format(new TelephoneFormatter(), "{0:N}",
0));
        Console.WriteLine(String.Format(new TelephoneFormatter(), "{0:N}",
911));
        Console.WriteLine(String.Format(new TelephoneFormatter(), "{0:N}",
8490216));
        Console.WriteLine(String.Format(new TelephoneFormatter(), "{0:N}",
4257884748));

        Console.WriteLine(String.Format(new TelephoneFormatter(), "{0:I}",
4257884748));
    }
}

```

The custom numeric format provider can be used only with the [String.Format\(IFormatProvider, String, Object\[\]\)](#) method. The other overloads of numeric formatting methods (such as [ToString](#)) that have a parameter of type [IFormatProvider](#) all pass the [IFormatProvider.GetFormat](#) implementation a [Type](#) object that represents the [NumberFormatInfo](#) type. In return, they expect the method to return a [NumberFormatInfo](#) object. If it does not, the custom numeric format provider is ignored, and the [NumberFormatInfo](#) object for the current culture is used in its place. In the example, the [TelephoneFormatter.GetFormat](#) method handles the possibility that it may

be inappropriately passed to a numeric formatting method by examining the method parameter and returning `null` if it represents a type other than [ICustomFormatter](#).

If a custom numeric format provider supports a set of format specifiers, make sure you provide a default behavior if no format specifier is supplied in the format item used in the [String.Format\(IFormatProvider, String, Object\[\]\)](#) method call. In the example, "N" is the default format specifier. This allows for a number to be converted to a formatted telephone number by providing an explicit format specifier. The following example illustrates such a method call.

C#

```
Console.WriteLine(String.Format(new TelephoneFormatter(), "{0:N}",  
4257884748));
```

But it also allows the conversion to occur if no format specifier is present. The following example illustrates such a method call.

C#

```
Console.WriteLine(String.Format(new TelephoneFormatter(), "{0}",  
4257884748));
```

If no default format specifier is defined, your implementation of the [ICustomFormatter.Format](#) method should include code such as the following so that .NET can provide formatting that your code does not support.

C#

```
if (arg is IFormattable)  
    s = ((IFormattable)arg).ToString(format, formatProvider);  
else if (arg != null)  
    s = arg.ToString();
```

In the case of this example, the method that implements [ICustomFormatter.Format](#) is intended to serve as a callback method for the [String.Format\(IFormatProvider, String, Object\[\]\)](#) method. Therefore, it examines the `formatProvider` parameter to determine whether it contains a reference to the current `TelephoneFormatter` object. However, the method can also be called directly from code. In that case, you can use the `formatProvider` parameter to provide a [CultureInfo](#) or [NumberFormatInfo](#) object that supplies culture-specific formatting information.

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

How to: Round-trip Date and time values

Article • 12/21/2022

In many applications, a date and time value is intended to unambiguously identify a single point in time. This article shows how to save and restore a [DateTime](#) value, a [DateTimeOffset](#) value, and a date and time value with time zone information so that the restored value identifies the same time as the saved value.

Round-trip a DateTime value

1. Convert the [DateTime](#) value to its string representation by calling the [DateTime.ToString\(String\)](#) method with the "o" format specifier.
2. Save the string representation of the [DateTime](#) value to a file, or pass it across a process, application domain, or machine boundary.
3. Retrieve the string that represents the [DateTime](#) value.
4. Call the [DateTime.Parse\(String, IFormatProvider, DateTimeStyles\)](#) method, and pass [DateTimeStyles.RoundtripKind](#) as the value of the `styles` parameter.

The following example illustrates how to round-trip a [DateTime](#) value.

C#

```
const string fileName = @"\DateFile.txt";

StreamWriter outFile = new StreamWriter(fileName);

// Save DateTime value.
DateTime dateToSave = DateTime.SpecifyKind(new DateTime(2008, 6, 12, 18, 45,
15),
                                            DateTimeKind.Local);
string? dateString = dateToSave.ToString("o");
Console.WriteLine("Converted {0} ({1}) to {2}.",
                  dateToSave.ToString(),
                  dateToSave.Kind.ToString(),
                  dateString);
outFile.WriteLine(dateString);
Console.WriteLine("Wrote {0} to {1}.", dateString, fileName);
outFile.Close();

// Restore DateTime value.
DateTime restoredDate;
```

```

using StreamReader inFile = new StreamReader(fileName);
dateString = inFile.ReadLine();

if (dateString is not null)
{
    restoredDate = DateTime.Parse(dateString, null,
    DateTimeStyles.RoundtripKind);
    Console.WriteLine("Read {0} ({2}) from {1}.", restoredDate.ToString(),
                      fileName,
    restoredDate.Kind.ToString());
}

// The example displays the following output:
//     Converted 6/12/2008 6:45:15 PM (Local) to 2008-06-12T18:45:15.0000000-
// 05:00.
//     Wrote 2008-06-12T18:45:15.0000000-05:00 to .\DateFile.txt.
//     Read 6/12/2008 6:45:15 PM (Local) from .\DateFile.txt.

```

When round-tripping a `DateTime` value, this technique successfully preserves the time for all local and universal times. For example, if a local `DateTime` value is saved on a system in the U.S. Pacific Standard Time zone and is restored on a system in the U.S. Central Standard Time zone, the restored date and time will be two hours later than the original time, which reflects the time difference between the two time zones. However, this technique is not necessarily accurate for unspecified times. All `DateTime` values whose `Kind` property is `Unspecified` are treated as if they are local times. If it's not a local time, the `DateTime` doesn't successfully identify the correct point in time. The workaround for this limitation is to tightly couple a date and time value with its time zone for the save and restore operation.

Round-trip a `DateTimeOffset` value

1. Convert the `DateTimeOffset` value to its string representation by calling the `DateTimeOffset.ToString(String)` method with the "o" format specifier.
2. Save the string representation of the `DateTimeOffset` value to a file, or pass it across a process, application domain, or machine boundary.
3. Retrieve the string that represents the `DateTimeOffset` value.
4. Call the `DateTimeOffset.Parse(String, IFormatProvider, DateTimeStyles)` method, and pass `DateTimeStyles.RoundtripKind` as the value of the `styles` parameter.

The following example illustrates how to round-trip a `DateTimeOffset` value.

C#

```
const string fileName = @".\DateOff.txt";

StreamWriter outFile = new StreamWriter(fileName);

// Save DateTime value.
DateTimeOffset dateToSave = new DateTimeOffset(2008, 6, 12, 18, 45, 15,
                                                new TimeSpan(7, 0, 0));
string? dateString = dateToSave.ToString("o");
Console.WriteLine("Converted {0} to {1}.", dateToSave.ToString(),
                  dateString);
outFile.WriteLine(dateString);
Console.WriteLine("Wrote {0} to {1}.", dateString, fileName);
outFile.Close();

// Restore DateTime value.
DateTimeOffset restoredDateOff;

using StreamReader inFile = new StreamReader(fileName);
dateString = inFile.ReadLine();

if (dateString is not null)
{
    restoredDateOff = DateTimeOffset.Parse(dateString, null,
                                            DateTimeStyles.RoundtripKind);
    Console.WriteLine("Read {0} from {1}.", restoredDateOff.ToString(),
                      fileName);
}

// The example displays the following output:
//     Converted 6/12/2008 6:45:15 PM +07:00 to 2008-06-
12T18:45:15.0000000+07:00.
//     Wrote 2008-06-12T18:45:15.0000000+07:00 to .\DateOff.txt.
//     Read 6/12/2008 6:45:15 PM +07:00 from .\DateOff.txt.
```

This technique always unambiguously identifies a [DateTimeOffset](#) value as a single point in time. The value can then be converted to Coordinated Universal Time (UTC) by calling the [DateTimeOffset.ToUniversalTime](#) method, or it can be converted to the time in a particular time zone by calling the [DateTimeOffset.ToOffset](#) or [TimeZoneInfo.ConvertTime\(DateTimeOffset, TimeZoneInfo\)](#) method. The major limitation of this technique is that date and time arithmetic, when performed on a [DateTimeOffset](#) value that represents the time in a particular time zone, may not produce accurate results for that time zone. This is because when a [DateTimeOffset](#) value is instantiated, it is disassociated from its time zone. Therefore, that time zone's adjustment rules can no longer be applied when you perform date and time calculations. You can work around this problem by defining a custom type that includes both a date and time value and its accompanying time zone.

Compile the code

These examples require that the following namespaces be imported with C# `using` directives or Visual Basic `Imports` statements:

- [System](#) (C# only)
- [System.Globalization](#)
- [System.IO](#)

See also

- [Choosing Between DateTime, DateTimeOffset, TimeSpan, and TimeZoneInfo](#)
- [Standard Date and Time Format Strings](#)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

How to: Display milliseconds in date and time values

Article • 10/04/2022

The default date and time formatting methods, such as [DateTime.ToString\(\)](#), include the hours, minutes, and seconds of a time value but exclude its milliseconds component. This article shows how to include a date and time's millisecond component in formatted date and time strings.

To display the millisecond component of a DateTime value

1. If you're working with the string representation of a date, convert it to a [DateTime](#) or a [DateTimeOffset](#) value by using the static [DateTime.Parse\(String\)](#) or [DateTimeOffset.Parse\(String\)](#) method.
2. To extract the string representation of a time's millisecond component, call the date and time value's [DateTime.ToString\(String\)](#) or [ToString](#) method, and pass the `fff` or `FFF` custom format pattern alone or with other custom format specifiers as the `format` parameter.

Tip

The [System.Globalization.NumberFormatInfo.NumberDecimalSeparator](#) property specifies the millisecond separator.

Example

The example displays the millisecond component of a [DateTime](#) and a [DateTimeOffset](#) value to the console, alone and included in a longer date and time string.

C#

```
using System.Globalization;
using System.Text.RegularExpressions;

string dateString = "7/16/2008 8:32:45.126 AM";

try
{
```

```

DateTime dateValue = DateTime.Parse(dateString);
DateTimeOffset dateOffsetValue = DateTimeOffset.Parse(dateString);

// Display Millisecond component alone.
Console.WriteLine("Millisecond component only: {0}",
    dateValue.ToString("fff"));
Console.WriteLine("Millisecond component only: {0}",
    dateOffsetValue.ToString("fff"));

// Display Millisecond component with full date and time.
Console.WriteLine("Date and Time with Milliseconds: {0}",
    dateValue.ToString("MM/dd/yyyy hh:mm:ss.fff tt"));
Console.WriteLine("Date and Time with Milliseconds: {0}",
    dateOffsetValue.ToString("MM/dd/yyyy hh:mm:ss.fff tt"));

string fullPattern = DateTimeFormatInfo.CurrentInfo.FullDateTimePattern;

// Create a format similar to .fff but based on the current culture.
string millisecondFormat = $""
{NumberFormatInfo.CurrentInfo.NumberDecimalSeparator}fff";

// Append millisecond pattern to current culture's full date time
pattern.
fullPattern = Regex.Replace(fullPattern, "(:ss|:s)",
"${$1${millisecondFormat}}");

// Display Millisecond component with modified full date and time
pattern.
Console.WriteLine("Modified full date time pattern: {0}",
    dateValue.ToString(fullPattern));
Console.WriteLine("Modified full date time pattern: {0}",
    dateOffsetValue.ToString(fullPattern));
}

catch (FormatException)
{
    Console.WriteLine("Unable to convert {0} to a date.", dateString);
}

// The example displays the following output if the current culture is en-
US:
//    Millisecond component only: 126
//    Millisecond component only: 126
//    Date and Time with Milliseconds: 07/16/2008 08:32:45.126 AM
//    Date and Time with Milliseconds: 07/16/2008 08:32:45.126 AM
//    Modified full date time pattern: Wednesday, July 16, 2008 8:32:45.126
AM
//    Modified full date time pattern: Wednesday, July 16, 2008 8:32:45.126
AM

```

The `fff` format pattern includes any trailing zeros in the millisecond value. The `FFF` format pattern suppresses them. The following example illustrates the difference:

C#

```
DateTime dateValue = new DateTime(2008, 7, 16, 8, 32, 45, 180);
Console.WriteLine(dateValue.ToString("fff"));
Console.WriteLine(dateValue.ToString("FFF"));
// The example displays the following output to the console:
//    180
//    18
```

A problem with defining a complete custom format specifier that includes the millisecond component of a date and time is that it defines a hard-coded format that might not correspond to the arrangement of time elements in the application's current culture. A better alternative is to retrieve one of the date and time display patterns defined by the current culture's [DateTimeFormatInfo](#) object and modify it to include milliseconds. The example also illustrates this approach. It retrieves the current culture's full date and time pattern from the [DateTimeFormatInfo.FullDateTimePattern](#) property and then inserts the custom pattern `fff` along with the current culture's millisecond separator. The example uses a regular expression to do this operation in a single method call.

You can also use a custom format specifier to display a fractional part of seconds other than milliseconds. For example, the `f` or `F` custom format specifier displays tenths of a second, the `ff` or `FF` custom format specifier displays hundredths of a second, and the `ffff` or `FFFF` custom format specifier displays ten-thousandths of a second. Fractional parts of a millisecond are truncated instead of rounded in the returned string. These format specifiers are used in the following example:

C#

```
DateTime dateValue = new DateTime(2008, 7, 16, 8, 32, 45, 180);
Console.WriteLine("{0} seconds", dateValue.ToString("s.f"));
Console.WriteLine("{0} seconds", dateValue.ToString("s.ff"));
Console.WriteLine("{0} seconds", dateValue.ToString("s.ffff"));
// The example displays the following output to the console:
//    45.1 seconds
//    45.18 seconds
//    45.1800 seconds
```

ⓘ Note

It's possible to display very small fractional units of a second, such as ten-thousandths of a second or hundred-thousandths of a second. However, these values might not be meaningful. The precision of a date and time value depends on the resolution of the operating system clock. For more information, see the API your operating system uses:

- Windows 7: `GetSystemTimeAsFileTime`
- Windows 8 and above: `GetSystemTimePreciseAsFileTime`
- Linux and macOS: `clock_gettime` ↗

See also

- [DateTimeFormatInfo](#)
- [Custom date and time format strings](#)

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

How to: Display Dates in Non-Gregorian Calendars

Article • 10/06/2022

The [DateTime](#) and [DateTimeOffset](#) types use the Gregorian calendar as their default calendar. This means that calling a date and time value's `ToString` method displays the string representation of that date and time in the Gregorian calendar, even if that date and time was created using another calendar. This is illustrated in the following example, which uses two different ways to create a date and time value with the Persian calendar, but still displays those date and time values in the Gregorian calendar when it calls the `ToString` method. This example reflects two commonly used but incorrect techniques for displaying the date in a particular calendar.

C#

```
PersianCalendar persianCal = new PersianCalendar();

DateTime persianDate = persianCal.ToDateTime(1387, 3, 18, 12, 0, 0, 0);
Console.WriteLine(persianDate.ToString());

persianDate = new DateTime(1387, 3, 18, persianCal);
Console.WriteLine(persianDate.ToString());
// The example displays the following output to the console:
//      6/7/2008 12:00:00 PM
//      6/7/2008 12:00:00 AM
```

Two different techniques can be used to display the date in a particular calendar. The first requires that the calendar be the default calendar for a particular culture. The second can be used with any calendar.

To display the date for a culture's default calendar

1. Instantiate a calendar object derived from the [Calendar](#) class that represents the calendar to be used.
2. Instantiate a [CultureInfo](#) object representing the culture whose formatting will be used to display the date.
3. Call the [Array.Exists](#) method to determine whether the calendar object is a member of the array returned by the [CultureInfo.OptionalCalendars](#) property. This indicates that the calendar can serve as the default calendar for the [CultureInfo](#) object. If it is

not a member of the array, follow the instructions in the "To Display the Date in Any Calendar" section.

4. Assign the calendar object to the [Calendar](#) property of the [DateTimeFormatInfo](#) object returned by the [CultureInfo.DateTimeFormat](#) property.

 **Note**

The [CultureInfo](#) class also has a [Calendar](#) property. However, it is read-only and constant; it does not change to reflect the new default calendar assigned to the [DateTimeFormatInfo.Calendar](#) property.

5. Call either the [ToString](#) or the [ToString](#) method, and pass it the [CultureInfo](#) object whose default calendar was modified in the previous step.

To display the date in any calendar

1. Instantiate a calendar object derived from the [Calendar](#) class that represents the calendar to be used.
2. Determine which date and time elements should appear in the string representation of the date and time value.
3. For each date and time element that you want to display, call the calendar object's [Get ...](#) method. The following methods are available:
 - [GetYear](#), to display the year in the appropriate calendar.
 - [GetMonth](#), to display the month in the appropriate calendar.
 - [GetDayOfMonth](#), to display the number of the day of the month in the appropriate calendar.
 - [GetHour](#), to display the hour of the day in the appropriate calendar.
 - [GetMinute](#), to display the minutes in the hour in the appropriate calendar.
 - [GetSecond](#), to display the seconds in the minute in the appropriate calendar.
 - [GetMilliseconds](#), to display the milliseconds in the second in the appropriate calendar.

Example

The example displays a date using two different calendars. It displays the date after defining the Hijri calendar as the default calendar for the ar-JO culture, and displays the date using the Persian calendar, which is not supported as an optional calendar by the fa-IR culture.

C#

```
using System;
using System.Globalization;

public class CalendarDates
{
    public static void Main()
    {
        HijriCalendar hijriCal = new HijriCalendar();
        CalendarUtility hijriUtil = new CalendarUtility(hijriCal);
        DateTime dateValue1 = new DateTime(1429, 6, 29, hijriCal);
        DateTimeOffset dateValue2 = new DateTimeOffset(dateValue1,
            TimeZoneInfo.Local.GetUtcOffset(dateValue1));
        CultureInfo jc = CultureInfo.CreateSpecificCulture("ar-JO");

        // Display the date using the Gregorian calendar.
        Console.WriteLine("Using the system default culture: {0}",
            dateValue1.ToString("d"));
        // Display the date using the ar-JO culture's original default
        // calendar.
        Console.WriteLine("Using the ar-JO culture's original default
calendar: {0}",
            dateValue1.ToString("d", jc));
        // Display the date using the Hijri calendar.
        Console.WriteLine("Using the ar-JO culture with Hijri as the default
calendar:");
        // Display a Date value.
        Console.WriteLine(hijriUtil.DisplayDate(dateValue1, jc));
        // Display a DateTimeOffset value.
        Console.WriteLine(hijriUtil.DisplayDate(dateValue2, jc));

        Console.WriteLine();

        PersianCalendar persianCal = new PersianCalendar();
        CalendarUtility persianUtil = new CalendarUtility(persianCal);
        CultureInfo ic = CultureInfo.CreateSpecificCulture("fa-IR");

        // Display the date using the ir-FA culture's default calendar.
        Console.WriteLine("Using the ir-FA culture's default calendar: {0}",
            dateValue1.ToString("d", ic));
        // Display a Date value.
        Console.WriteLine(persianUtil.DisplayDate(dateValue1, ic));
        // Display a DateTimeOffset value.
        Console.WriteLine(persianUtil.DisplayDate(dateValue2, ic));
    }
}
```

```
public class CalendarUtility
{
    private Calendar thisCalendar;
    private CultureInfo targetCulture;

    public CalendarUtility(Calendar cal)
    {
        this.thisCalendar = cal;
    }

    private bool CalendarExists(CultureInfo culture)
    {
        this.targetCulture = culture;
        return Array.Exists(this.targetCulture.OptionalCalendars,
                            this.HasSameName);
    }

    private bool HasSameName(Calendar cal)
    {
        if (cal.ToString() == thisCalendar.ToString())
            return true;
        else
            return false;
    }

    public string DisplayDate(DateTime dateToDisplay, CultureInfo culture)
    {
        DateTimeOffset displayOffsetDate = dateToDisplay;
        return DisplayDate(displayOffsetDate, culture);
    }

    public string DisplayDate(DateTimeOffset dateToDisplay,
                             CultureInfo culture)
    {
        string specifier = "yyyy/MM/dd";

        if (this.CalendarExists(culture))
        {
            Console.WriteLine("Displaying date in supported {0} calendar...",
                              this.thisCalendar.GetType().Name);
            culture.DateTimeFormat.Calendar = this.thisCalendar;
            return dateToDisplay.ToString(specifier, culture);
        }
        else
        {
            Console.WriteLine("Displaying date in unsupported {0} calendar...",
                              thisCalendar.GetType().Name);

            string separator = targetCulture.DateTimeFormat.DateSeparator;

            return
thisCalendar.GetYear(dateToDisplay.DateTime).ToString("0000") +
                    separator +
                    thisCalendar.GetMonth(dateToDisplay.DateTime).ToString("00")
        }
    }
}
```

```

+
separator +

thisCalendar.GetDayOfMonth(dateToDisplay.DateTime).ToString("00");
    }
}
// The example displays the following output to the console:
//      Using the system default culture: 7/3/2008
//      Using the ar-JO culture's original default calendar: 03/07/2008
//      Using the ar-JO culture with Hijri as the default calendar:
//          Displaying date in supported HijriCalendar calendar...
//          1429/06/29
//          Displaying date in supported HijriCalendar calendar...
//          1429/06/29
//
//      Using the ir-FA culture's default calendar: 7/3/2008
//      Displaying date in unsupported PersianCalendar calendar...
//      1387/04/13
//      Displaying date in unsupported PersianCalendar calendar...
//      1387/04/13

```

Each [CultureInfo](#) object can support one or more calendars, which are indicated by the [OptionalCalendars](#) property. One of these is designated as the culture's default calendar and is returned by the read-only [CultureInfo.Calendar](#) property. Another of the optional calendars can be designated as the default by assigning a [Calendar](#) object that represents that calendar to the [DateTimeFormatInfo.Calendar](#) property returned by the [CultureInfo.DateFormat](#) property. However, some calendars, such as the Persian calendar represented by the [PersianCalendar](#) class, do not serve as optional calendars for any culture.

The example defines a reusable calendar utility class, [CalendarUtility](#), to handle many of the details of generating the string representation of a date using a particular calendar. The [CalendarUtility](#) class has the following members:

- A parameterized constructor whose single parameter is a [Calendar](#) object in which a date is to be represented. This is assigned to a private field of the class.
- [CalendarExists](#), a private method that returns a Boolean value indicating whether the calendar represented by the [CalendarUtility](#) object is supported by the [CultureInfo](#) object that is passed to the method as a parameter. The method wraps a call to the [Array.Exists](#) method, to which it passes the [CultureInfo.OptionalCalendars](#) array.
- [HasSameName](#), a private method assigned to the [Predicate<T>](#) delegate that is passed as a parameter to the [Array.Exists](#) method. Each member of the array is passed to the method until the method returns [true](#). The method determines

whether the name of an optional calendar is the same as the calendar represented by the `CalendarUtility` object.

- `DisplayDate`, an overloaded public method that is passed two parameters: either a `DateTime` or `DateTimeOffset` value to express in the calendar represented by the `CalendarUtility` object; and the culture whose formatting rules are to be used. Its behavior in returning the string representation of a date depends on whether the target calendar is supported by the culture whose formatting rules are to be used.

Regardless of the calendar used to create a `DateTime` or `DateTimeOffset` value in this example, that value is typically expressed as a Gregorian date. This is because the `DateTime` and `DateTimeOffset` types do not preserve any calendar information.

Internally, they are represented as the number of ticks that have elapsed since midnight of January 1, 0001. The interpretation of that number depends on the calendar. For most cultures, the default calendar is the Gregorian calendar.

 **Collaborate with us on GitHub**

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback [here](#).

 [Open a documentation issue](#)

 [Provide product feedback](#)

Character encoding in .NET

Article • 02/14/2023

This article provides an introduction to character encoding systems that are used by .NET. The article explains how the [String](#), [Char](#), [Rune](#), and [StringInfo](#) types work with Unicode, UTF-16, and UTF-8.

The term *character* is used here in the general sense of *what a reader perceives as a single display element*. Common examples are the letter "a", the symbol "@", and the emoji "🐂". Sometimes what looks like one character is actually composed of multiple independent display elements, as the section on [grapheme clusters](#) explains.

The string and char types

An instance of the [string](#) class represents some text. A `string` is logically a sequence of 16-bit values, each of which is an instance of the [char](#) struct. The `string.Length` property returns the number of `char` instances in the `string` instance.

The following sample function prints out the values in hexadecimal notation of all the `char` instances in a `string`:

```
C#  
  
void PrintChars(string s)  
{  
    Console.WriteLine($"\"{s}\".Length = {s.Length}");  
    for (int i = 0; i < s.Length; i++)  
    {  
        Console.WriteLine($"s[{i}] = '{s[i]}' ('\\u{(int)s[i]:x4}')");  
    }  
    Console.WriteLine();  
}
```

Pass the string "Hello" to this function, and you get the following output:

```
C#  
  
PrintChars("Hello");
```

Output

```
"Hello".Length = 5  
s[0] = 'H' ('\u0048')
```

```
s[1] = 'e' ('\u0065')
s[2] = 'l' ('\u006c')
s[3] = 'l' ('\u006c')
s[4] = 'o' ('\u006f')
```

Each character is represented by a single `char` value. That pattern holds true for most of the world's languages. For example, here's the output for two Chinese characters that sound like *nǐ hǎo* and mean *Hello*:

C#

```
PrintChars("你好");
```

Output

```
"你好".Length = 2
s[0] = '你' ('\u4f60')
s[1] = '好' ('\u597d')
```

However, for some languages and for some symbols and emoji, it takes two `char` instances to represent a single character. For example, compare the characters and `char` instances in the word that means *Osage* in the Osage language:

C#

```
PrintChars("ଓସାଜେ ଓସାଜେ");
```

Output

```
"ଓସାଜେ ଓସାଜେ".Length = 17
s[0] = 'ଡ' ('\ud801')
s[1] = 'ଡ' ('\udccf')
s[2] = 'ଡ' ('\ud801')
s[3] = 'ଡ' ('\udcd8')
s[4] = 'ଡ' ('\ud801')
s[5] = 'ଡ' ('\udcfb')
s[6] = 'ଡ' ('\ud801')
s[7] = 'ଡ' ('\udcd8')
s[8] = 'ଡ' ('\ud801')
s[9] = 'ଡ' ('\udcfb')
s[10] = 'ଡ' ('\ud801')
s[11] = 'ଡ' ('\udcdf')
s[12] = ' ' ('\u0020')
s[13] = 'ଡ' ('\ud801')
s[14] = 'ଡ' ('\udcbb')
```

```
s[15] = '߻' ('\\ud801')
s[16] = '߻' ('\\udcdf')
```

In the preceding example, each character except the space is represented by two `char` instances.

A single Unicode emoji is also represented by two `char`s, as seen in the following example showing an ox emoji:

Output

```
"🐂".Length = 2
s[0] = '߻' ('\\ud83d')
s[1] = '߻' ('\\udc02')
```

These examples show that the value of `string.Length`, which indicates the number of `char` instances, doesn't necessarily indicate the number of displayed characters. A single `char` instance by itself doesn't necessarily represent a character.

The `char` pairs that map to a single character are called *surrogate pairs*. To understand how they work, you need to understand Unicode and UTF-16 encoding.

Unicode code points

Unicode is an international encoding standard for use on various platforms and with various languages and scripts.

The Unicode Standard defines over 1.1 million [code points](#). A code point is an integer value that can range from 0 to `U+10FFFF` (decimal 1,114,111). Some code points are assigned to letters, symbols, or emoji. Others are assigned to actions that control how text or characters are displayed, such as advance to a new line. Many code points are not yet assigned.

Here are some examples of code point assignments, with links to Unicode charts in which they appear:

| Decimal | Hex | Example | Description |
|---------|---------------------|---------|--|
| 10 | <code>U+000A</code> | N/A | LINE FEED |
| 97 | <code>U+0061</code> | a | LATIN SMALL LETTER A |
| 562 | <code>U+0232</code> | Ŷ | LATIN CAPITAL LETTER Y WITH MACRON |

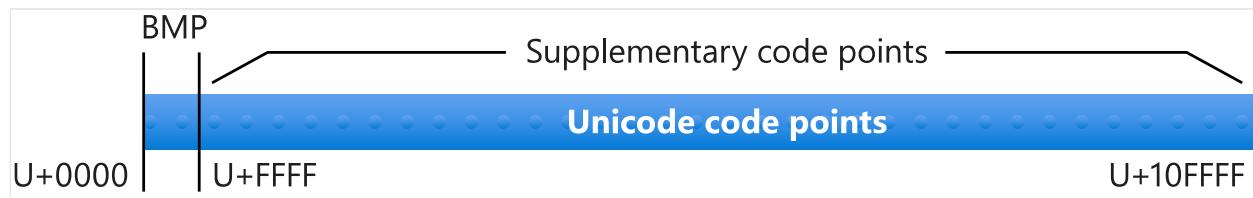
| Decimal | Hex | Example | Description |
|---------|---------|---------|-------------------------------|
| 68,675 | U+10C43 | ꝧ | OLD TURKIC LETTER ORKHON AT ↗ |
| 127,801 | U+1F339 | 🌹 | ROSE emoji ↗ |

Code points are customarily referred to by using the syntax `U+xxxx`, where `xxxx` is the hex-encoded integer value.

Within the full range of code points there are two subranges:

- The **Basic Multilingual Plane (BMP)** in the range `U+0000..U+FFFF`. This 16-bit range provides 65,536 code points, enough to cover the majority of the world's writing systems.
- **Supplementary code points** in the range `U+10000..U+10FFFF`. This 21-bit range provides more than a million additional code points that can be used for less well-known languages and other purposes such as emojis.

The following diagram illustrates the relationship between the BMP and the supplementary code points.



UTF-16 code units

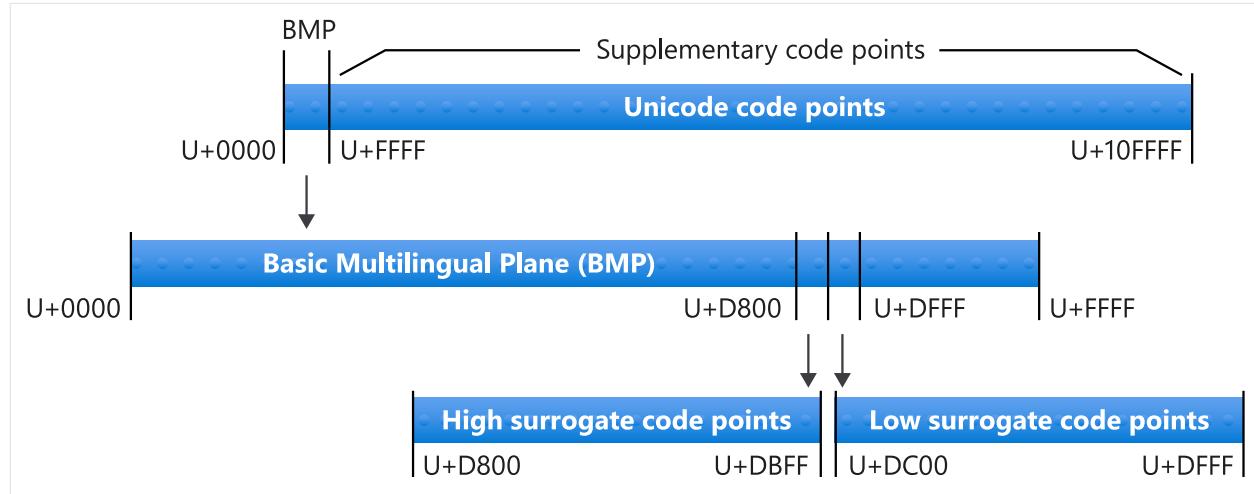
16-bit Unicode Transformation Format ([UTF-16](#)) is a character encoding system that uses 16-bit *code units* to represent Unicode code points. .NET uses UTF-16 to encode the text in a `string`. A `char` instance represents a 16-bit code unit.

A single 16-bit code unit can represent any code point in the 16-bit range of the Basic Multilingual Plane. But for a code point in the supplementary range, two `char` instances are needed.

Surrogate pairs

The translation of two 16-bit values to a single 21-bit value is facilitated by a special range called the *surrogate code points*, from `U+D800` to `U+DFFF` (decimal 55,296 to 57,343), inclusive.

The following diagram illustrates the relationship between the BMP and the surrogate code points.



When a *high surrogate* code point (U+D800..U+DBFF) is immediately followed by a *low surrogate* code point (U+DC00..U+DFFF), the pair is interpreted as a supplementary code point by using the following formula:

```
code point = 0x10000 +
    ((high surrogate code point - 0xD800) * 0x0400) +
    (low surrogate code point - 0xDC00)
```

Here's the same formula using decimal notation:

```
code point = 65,536 +
    ((high surrogate code point - 55,296) * 1,024) +
    (low surrogate code point - 56,320)
```

A *high surrogate* code point doesn't have a higher number value than a *low surrogate* code point. The high surrogate code point is called "high" because it's used to calculate the higher-order 10 bits of a 20-bit code point range. The low surrogate code point is used to calculate the lower-order 10 bits.

For example, the actual code point that corresponds to the surrogate pair `0xD83C` and `0xDF39` is computed as follows:

```
actual = 0x10000 + ((0xD83C - 0xD800) * 0x0400) + (0xDF39 - 0xDC00)
        = 0x10000 + (0x003C * 0x0400) + 0x0339
```

```
= 0x10000 + 0xF000 + 0x0339  
= 0x1F339
```

Here's the same calculation using decimal notation:

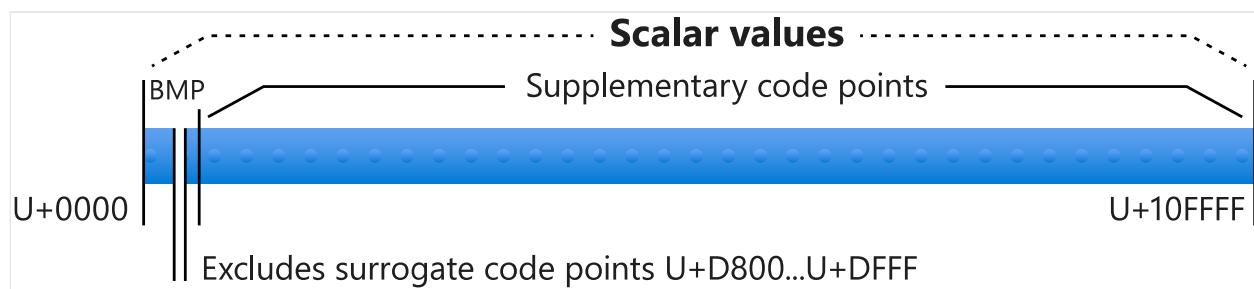
```
actual = 65,536 + ((55,356 - 55,296) * 1,024) + (57,145 - 56320)  
= 65,536 + (60 * 1,024) + 825  
= 65,536 + 61,440 + 825  
= 127,801
```

The preceding example demonstrates that `"\ud83c\udf39"` is the UTF-16 encoding of the `U+1F339 ROSE ('🌹')` code point mentioned earlier.

Unicode scalar values

The term [Unicode scalar value](#) refers to all code points other than the surrogate code points. In other words, a scalar value is any code point that is assigned a character or can be assigned a character in the future. "Character" here refers to anything that can be assigned to a code point, which includes such things as actions that control how text or characters are displayed.

The following diagram illustrates the scalar value code points.



The Rune type as a scalar value

Beginning with .NET Core 3.0, the `System.Text.Rune` type represents a Unicode scalar value. `Rune` is not available in .NET Core 2.x or .NET Framework 4.x.

The `Rune` constructors validate that the resulting instance is a valid Unicode scalar value, otherwise they throw an exception. The following example shows code that successfully instantiates `Rune` instances because the input represents valid scalar values:

```
C#
```

```
Rune a = new Rune('a');
Rune b = new Rune(0x0061);
Rune c = new Rune('\u0061');
Rune d = new Rune(0x10421);
Rune e = new Rune('\ud801', '\udc21');
```

The following example throws an exception because the code point is in the surrogate range and isn't part of a surrogate pair:

C#

```
Rune f = new Rune('\ud801');
```

The following example throws an exception because the code point is beyond the supplementary range:

C#

```
Rune g = new Rune(0x12345678);
```

Rune usage example: changing letter case

An API that takes a `char` and assumes it is working with a code point that is a scalar value doesn't work correctly if the `char` is from a surrogate pair. For example, consider the following method that calls `Char.ToUpperInvariant` on each char in a string:

C#

```
// THE FOLLOWING METHOD SHOWS INCORRECT CODE.
// DO NOT DO THIS IN A PRODUCTION APPLICATION.
static string ConvertToUpperBadExample(string input)
{
    StringBuilder builder = new StringBuilder(input.Length);
    for (int i = 0; i < input.Length; i++) /* or 'foreach' */
    {
        builder.Append(char.ToUpperInvariant(input[i]));
    }
    return builder.ToString();
}
```

If the `input` string contains the lowercase Deseret letter `er` (⊛), this code won't convert it to uppercase (⊛). The code calls `char.ToUpperInvariant` separately on each surrogate code point, `U+D801` and `U+DC49`. But `U+D801` doesn't have enough information by itself to identify it as a lowercase letter, so `char.ToUpperInvariant` leaves it alone. And it

handles `U+DC49` the same way. The result is that lowercase 'ɸ' in the `input` string doesn't get converted to uppercase 'Φ'.

Here are two options for correctly converting a string to uppercase:

- Call [String.ToUpperInvariant](#) on the input string rather than iterating `char`-by-`char`. The `string.ToUpperInvariant` method has access to both parts of each surrogate pair, so it can handle all Unicode code points correctly.
- Iterate through the Unicode scalar values as `Rune` instances instead of `char` instances, as shown in the following example. Since a `Rune` instance is a valid Unicode scalar value, it can be passed to APIs that expect to operate on a scalar value. For example, calling [Rune.ToUpperInvariant](#) as shown in the following example gives correct results:

C#

```
static string ConvertToUpper(string input)
{
    StringBuilder builder = new StringBuilder(input.Length);
    foreach (Rune rune in input.EnumerateRunes())
    {
        builder.Append(Rune.ToUpperInvariant(rune));
    }
    return builder.ToString();
}
```

Other Rune APIs

The `Rune` type exposes analogs of many of the `char` APIs. For example, the following methods mirror static APIs on the `char` type:

- [Rune.IsLetter](#)
- [Rune.IsWhiteSpace](#)
- [Rune.IsLetterOrDigit](#)
- [Rune.GetUnicodeCategory](#)

To get the raw scalar value from a `Rune` instance, use the [Rune.Value](#) property.

To convert a `Rune` instance back to a sequence of `char`s, use [Rune.ToString](#) or the [Rune.EncodeToUtf16](#) method.

Since any Unicode scalar value is representable by a single `char` or by a surrogate pair, any `Rune` instance can be represented by at most 2 `char` instances. Use

`Rune.Utf16SequenceLength` to see how many `char` instances are required to represent a `Rune` instance.

For more information about the .NET `Rune` type, see the [Rune API reference](#).

Grapheme clusters

What looks like one character might result from a combination of multiple code points, so a more descriptive term that is often used in place of "character" is [grapheme cluster](#). The equivalent term in .NET is [text element](#).

Consider the `string` instances "a", "á", "á", and "". If your operating system handles them as specified by the Unicode standard, each of these `string` instances appears as a single text element or grapheme cluster. But the last two are represented by more than one scalar value code point.

- The string "a" is represented by one scalar value and contains one `char` instance.
 - `U+0061 LATIN SMALL LETTER A`
- The string "á" is represented by one scalar value and contains one `char` instance.
 - `U+00E1 LATIN SMALL LETTER A WITH ACUTE`
- The string "á" looks the same as "á" but is represented by two scalar values and contains two `char` instances.
 - `U+0061 LATIN SMALL LETTER A`
 - `U+0301 COMBINING ACUTE ACCENT`
- Finally, the string "" is represented by four scalar values and contains seven `char` instances.
 - `U+1F469 WOMAN` (supplementary range, requires a surrogate pair)
 - `U+1F3FD EMOJI MODIFIER FITZPATRICK TYPE-4` (supplementary range, requires a surrogate pair)
 - `U+200D ZERO WIDTH JOINER`
 - `U+1F692 FIRE ENGINE` (supplementary range, requires a surrogate pair)

In some of the preceding examples - such as the combining accent modifier or the skin tone modifier - the code point does not display as a standalone element on the screen. Rather, it serves to modify the appearance of a text element that came before it. These examples show that it might take multiple scalar values to make up what we think of as a single "character," or "grapheme cluster."

To enumerate the grapheme clusters of a `string`, use the `StringInfo` class as shown in the following example. If you're familiar with Swift, the .NET `StringInfo` type is conceptually similar to Swift's `character type` ↗.

Example: count char, Rune, and text element instances

In .NET APIs, a grapheme cluster is called a *text element*. The following method demonstrates the differences between `char`, `Rune`, and text element instances in a `string`:

C#

```
static void PrintTextElementCount(string s)
{
    Console.WriteLine(s);
    Console.WriteLine($"Number of chars: {s.Length}");
    Console.WriteLine($"Number of runes: {s.EnumerateRunes().Count()}");

    TextElementEnumerator enumerator =
StringInfo.GetTextElementEnumerator(s);

    int textElementCount = 0;
    while (enumerator.MoveNext())
    {
        textElementCount++;
    }

    Console.WriteLine($"Number of text elements: {textElementCount}");
}
```

C#

```
PrintTextElementCount("a");
// Number of chars: 1
// Number of runes: 1
// Number of text elements: 1

PrintTextElementCount("á");
// Number of chars: 2
// Number of runes: 2
// Number of text elements: 1

PrintTextElementCount("𠮷𠮶𠮶");
// Number of chars: 7
// Number of runes: 4
// Number of text elements: 1
```

If you run this code in .NET Framework or .NET Core 3.1 or earlier, the text element count for the emoji shows 4. That is due to a bug in the `StringInfo` class that is fixed in .NET 5.

Example: splitting string instances

When splitting `string` instances, avoid splitting surrogate pairs and grapheme clusters. Consider the following example of incorrect code, which intends to insert line breaks every 10 characters in a string:

C#

```
// THE FOLLOWING METHOD SHOWS INCORRECT CODE.  
// DO NOT DO THIS IN A PRODUCTION APPLICATION.  
static string InsertNewlinesEveryTencharsBadExample(string input)  
{  
    StringBuilder builder = new StringBuilder();  
  
    // First, append chunks in multiples of 10 chars  
    // followed by a newline.  
    int i = 0;  
    for (; i < input.Length - 10; i += 10)  
    {  
        builder.Append(input, i, 10);  
        builder.AppendLine(); // newline  
    }  
  
    // Then append any leftover data followed by  
    // a final newline.  
    builder.Append(input, i, input.Length - i);  
    builder.AppendLine(); // newline  
  
    return builder.ToString();  
}
```

Because this code enumerates `char` instances, a surrogate pair that happens to straddle a 10-`char` boundary will be split and a newline injected between them. This insertion introduces data corruption, because surrogate code points are meaningful only as pairs.

The potential for data corruption isn't eliminated if you enumerate `Rune` instances (scalar values) instead of `char` instances. A set of `Rune` instances might make up a grapheme cluster that straddles a 10-`char` boundary. If the grapheme cluster set is split up, it can't be interpreted correctly.

A better approach is to break the string by counting grapheme clusters, or text elements, as in the following example:

C#

```
static string InsertNewlinesEveryTenTextElements(string input)
{
    StringBuilder builder = new StringBuilder();

    // Append chunks in multiples of 10 chars

    TextElementEnumerator enumerator =
    StringInfo.GetTextElementEnumerator(input);

    int textElementCount = 1;
    while (enumerator.MoveNext())
    {
        builder.Append(enumerator.Current);
        if (textElementCount % 10 == 0 && textElementCount > 0)
        {
            builder.AppendLine(); // newline
        }
        textElementCount++;
    }

    // Add a final newline.
    builder.AppendLine(); // newline
    return builder.ToString();
}
```

As noted earlier, prior to .NET 5, the `StringInfo` class had a bug causing some grapheme clusters to be handled incorrectly.

UTF-8 and UTF-32

The preceding sections focused on UTF-16 because that's what .NET uses to encode `string` instances. There are other encoding systems for Unicode - [UTF-8 ↗](#) and [UTF-32 ↗](#). These encodings use 8-bit code units and 32-bit code units, respectively.

Like UTF-16, UTF-8 requires multiple code units to represent some Unicode scalar values. UTF-32 can represent any scalar value in a single 32-bit code unit.

Here are some examples showing how the same Unicode code point is represented in each of these three Unicode encoding systems:

| | |
|---|---------------------------------------|
| Scalar: U+0061 LATIN SMALL LETTER A ('a') | |
| UTF-8 : [61] | (1x 8-bit code unit = 8 bits total) |
| UTF-16: [0061] | (1x 16-bit code unit = 16 bits total) |

```
UTF-32: [ 00000061 ]      (1x 32-bit code unit = 32 bits total)

Scalar: U+0429 CYRILLIC CAPITAL LETTER SHCHA ('Щ')
UTF-8 : [ D0 A9 ]          (2x 8-bit code units = 16 bits total)
UTF-16: [ 0429 ]           (1x 16-bit code unit = 16 bits total)
UTF-32: [ 00000429 ]       (1x 32-bit code unit = 32 bits total)

Scalar: U+A992 JAVANESE LETTER GA ('া')
UTF-8 : [ EA A6 92 ]       (3x 8-bit code units = 24 bits total)
UTF-16: [ A992 ]           (1x 16-bit code unit = 16 bits total)
UTF-32: [ 0000A992 ]       (1x 32-bit code unit = 32 bits total)

Scalar: U+104CC OSAGE CAPITAL LETTER TSHA ('ଡ')
UTF-8 : [ F0 90 93 8C ]    (4x 8-bit code units = 32 bits total)
UTF-16: [ D801 DCCC ]      (2x 16-bit code units = 32 bits total)
UTF-32: [ 000104CC ]       (1x 32-bit code unit = 32 bits total)
```

As noted earlier, a single UTF-16 code unit from a [surrogate pair](#) is meaningless by itself. In the same way, a single UTF-8 code unit is meaningless by itself if it's in a sequence of two, three, or four used to calculate a scalar value.

ⓘ Note

Beginning with C# 11, you can represent UTF-8 string literals using the "u8" suffix on a literal string. For more information on UTF-8 string literals, see the "string literals" section of the article on [built in reference types](#) in the C# Guide.

Endianness

In .NET, the UTF-16 code units of a string are stored in contiguous memory as a sequence of 16-bit integers (`char` instances). The bits of individual code units are laid out according to the [endianness](#) of the current architecture.

On a little-endian architecture, the string consisting of the UTF-16 code points `[D801 DCCC]` would be laid out in memory as the bytes `[0x01, 0xD8, 0xCC, 0xDC]`. On a big-endian architecture that same string would be laid out in memory as the bytes `[0x01, 0xDC, 0xCC]`.

Computer systems that communicate with each other must agree on the representation of data crossing the wire. Most network protocols use UTF-8 as a standard when transmitting text, partly to avoid issues that might result from a big-endian machine communicating with a little-endian machine. The string consisting of the UTF-8 code points `[F0 90 93 8C]` will always be represented as the bytes `[0xF0, 0x90, 0x93, 0x8C]` regardless of endianness.

To use UTF-8 for transmitting text, .NET applications often use code like the following example:

```
C#
```

```
string stringToWrite = GetString();
byte[] stringAsUtf8Bytes = Encoding.UTF8.GetBytes(stringToWrite);
await outputStream.WriteAsync(stringAsUtf8Bytes, 0,
stringAsUtf8Bytes.Length);
```

In the preceding example, the method `Encoding.UTF8.GetBytes` decodes the UTF-16 `string` back into a series of Unicode scalar values, then it re-encodes those scalar values into UTF-8 and places the resulting sequence into a `byte` array. The method `Encoding.UTF8.GetString` performs the opposite transformation, converting a UTF-8 `byte` array to a UTF-16 `string`.

⚠ Warning

Since UTF-8 is commonplace on the internet, it may be tempting to read raw bytes from the wire and to treat the data as if it were UTF-8. However, you should validate that it is indeed well-formed. A malicious client might submit ill-formed UTF-8 to your service. If you operate on that data as if it were well-formed, it could cause errors or security holes in your application. To validate UTF-8 data, you can use a method like `Encoding.UTF8.GetString`, which will perform validation while converting the incoming data to a `string`.

Well-formed encoding

A well-formed Unicode encoding is a string of code units that can be decoded unambiguously and without error into a sequence of Unicode scalar values. Well-formed data can be transcoded freely back and forth between UTF-8, UTF-16, and UTF-32.

The question of whether an encoding sequence is well-formed or not is unrelated to the endianness of a machine's architecture. An ill-formed UTF-8 sequence is ill-formed in the same way on both big-endian and little-endian machines.

Here are some examples of ill-formed encodings:

- In UTF-8, the sequence `[6C C2 61]` is ill-formed because `C2` cannot be followed by `61`.

- In UTF-16, the sequence [DC00 DD00] (or, in C#, the string "\udc00\udd00") is ill-formed because the low surrogate DC00 cannot be followed by another low surrogate DD00.
- In UTF-32, the sequence [0011ABCD] is ill-formed because 0011ABCD is outside the range of Unicode scalar values.

In .NET, `string` instances almost always contain well-formed UTF-16 data, but that isn't guaranteed. The following examples show valid C# code that creates ill-formed UTF-16 data in `string` instances.

- An ill-formed literal:

```
C#
```

```
const string s = "\ud800";
```

- A substring that splits up a surrogate pair:

```
C#
```

```
string x = "\ud83e\udd70"; // "𩿱"
string y = x.Substring(1, 1); // "\udd70" standalone low surrogate
```

APIs like `Encoding.UTF8.GetString` never return ill-formed `string` instances.

`Encoding.GetString` and `Encoding.GetBytes` methods detect ill-formed sequences in the input and perform character substitution when generating the output. For example, if `Encoding.ASCII.GetString(byte[])` sees a non-ASCII byte in the input (outside the range U+0000..U+007F), it inserts a '?' into the returned `string` instance.

`Encoding.UTF8.GetString(byte[])` replaces ill-formed UTF-8 sequences with `U+FFFD` REPLACEMENT CHARACTER ('ߏ') in the returned `string` instance. For more information, see the [Unicode Standard](#), Sections 5.22 and 3.9.

The built-in `Encoding` classes can also be configured to throw an exception rather than perform character substitution when ill-formed sequences are seen. This approach is often used in security-sensitive applications where character substitution might not be acceptable.

```
C#
```

```
byte[] utf8Bytes = ReadFromNetwork();
UTF8Encoding encoding = new UTF8Encoding(encoderShouldEmitUTF8Identifier:
    false, throwOnInvalidBytes: true);
```

```
string asString = encoding.GetString(utf8Bytes); // will throw if  
'utf8Bytes' is ill-formed
```

For information about how to use the built-in `Encoding` classes, see [How to use character encoding classes in .NET](#).

See also

- [String](#)
- [Char](#)
- [Rune](#)
- [Globalization and localization](#)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

How to use character encoding classes in .NET

Article • 09/15/2021

This article explains how to use the classes that .NET provides for encoding and decoding text by using various encoding schemes. The instructions assume you have read [Introduction to character encoding in .NET](#).

Encoders and decoders

.NET provides encoding classes that encode and decode text by using various encoding systems. For example, the [UTF8Encoding](#) class describes the rules for encoding to, and decoding from, UTF-8. .NET uses UTF-16 encoding (represented by the [UnicodeEncoding](#) class) for `string` instances. Encoders and decoders are available for other encoding schemes.

Encoding and decoding can also include validation. For example, the [UnicodeEncoding](#) class checks all `char` instances in the surrogate range to make sure they're in valid surrogate pairs. A fallback strategy determines how an encoder handles invalid characters or how a decoder handles invalid bytes.

Warning

.NET encoding classes provide a way to store and convert character data. They should not be used to store binary data in string form. Depending on the encoding used, converting binary data to string format with the encoding classes can introduce unexpected behavior and produce inaccurate or corrupted data. To convert binary data to a string form, use the [Convert.ToString](#) method.

All character encoding classes in .NET inherit from the [System.Text.Encoding](#) class, which is an abstract class that defines the functionality common to all character encodings. To access the individual encoding objects implemented in .NET, do the following:

- Use the static properties of the [Encoding](#) class, which return objects that represent the standard character encodings available in .NET (ASCII, UTF-7, UTF-8, UTF-16, and UTF-32). For example, the [Encoding.Unicode](#) property returns a [UnicodeEncoding](#) object. Each object uses replacement fallback to handle strings that it cannot encode and bytes that it cannot decode. For more information, see [Replacement fallback](#).

- Call the encoding's class constructor. Objects for the ASCII, UTF-7, UTF-8, UTF-16, and UTF-32 encodings can be instantiated in this way. By default, each object uses replacement fallback to handle strings that it cannot encode and bytes that it cannot decode, but you can specify that an exception should be thrown instead. For more information, see [Replacement fallback](#) and [Exception fallback](#).
- Call the [Encoding\(Int32\)](#) constructor and pass it an integer that represents the encoding. Standard encoding objects use replacement fallback, and code page and double-byte character set (DBCS) encoding objects use best-fit fallback to handle strings that they cannot encode and bytes that they cannot decode. For more information, see [Best-fit fallback](#).
- Call the [Encoding.GetEncoding](#) method, which returns any standard, code page, or DBCS encoding available in .NET. Overloads let you specify a fallback object for both the encoder and the decoder.

You can retrieve information about all the encodings available in .NET by calling the [Encoding.GetEncodings](#) method. .NET supports the character encoding schemes listed in the following table.

| Encoding class | Description |
|------------------------|---|
| ASCII | Encodes a limited range of characters by using the lower seven bits of a byte. Because this encoding only supports character values from <code>U+0000</code> through <code>U+007F</code> , in most cases it is inadequate for internationalized applications. |
| UTF-7 | Represents characters as sequences of 7-bit ASCII characters. Non-ASCII Unicode characters are represented by an escape sequence of ASCII characters. UTF-7 supports protocols such as email and newsgroup. However, UTF-7 is not particularly secure or robust. In some cases, changing one bit can radically alter the interpretation of an entire UTF-7 string. In other cases, different UTF-7 strings can encode the same text. For sequences that include non-ASCII characters, UTF-7 requires more space than UTF-8, and encoding/decoding is slower. Consequently, you should use UTF-8 instead of UTF-7 if possible. |
| UTF-8 | Represents each Unicode code point as a sequence of one to four bytes. UTF-8 supports 8-bit data sizes and works well with many existing operating systems. For the ASCII range of characters, UTF-8 is identical to ASCII encoding and allows a broader set of characters. However, for Chinese-Japanese-Korean (CJK) scripts, UTF-8 can require three bytes for each character, and can cause larger data sizes than UTF-16. Sometimes the amount of ASCII data, such as HTML tags, justifies the increased size for the CJK range. |
| UTF-16 | Represents each Unicode code point as a sequence of one or two 16-bit integers. Most common Unicode characters require only one UTF-16 code point, although Unicode supplementary characters (<code>U+10000</code> and greater) require two UTF-16 |

| Encoding class | Description |
|--|---|
| | surrogate code points. Both little-endian and big-endian byte orders are supported. UTF-16 encoding is used by the common language runtime to represent Char and String values, and it is used by the Windows operating system to represent WCHAR values. |
| UTF-32 | Represents each Unicode code point as a 32-bit integer. Both little-endian and big-endian byte orders are supported. UTF-32 encoding is used when applications want to avoid the surrogate code point behavior of UTF-16 encoding on operating systems for which encoded space is too important. Single glyphs rendered on a display can still be encoded with more than one UTF-32 character. |
| ANSI/ISO encoding | Provides support for a variety of code pages. On Windows operating systems, code pages are used to support a specific language or group of languages. For a table that lists the code pages supported by .NET, see the Encoding class. You can retrieve an encoding object for a particular code page by calling the Encoding.GetEncoding(Int32) method. A code page contains 256 code points and is zero-based. In most code pages, code points 0 through 127 represent the ASCII character set, and code points 128 through 255 differ significantly between code pages. For example, code page 1252 provides the characters for Latin writing systems, including English, German, and French. The last 128 code points in code page 1252 contain the accent characters. Code page 1253 provides character codes that are required in the Greek writing system. The last 128 code points in code page 1253 contain the Greek characters. As a result, an application that relies on ANSI code pages cannot store Greek and German in the same text stream unless it includes an identifier that indicates the referenced code page. |
| Double-byte character set (DBCS) encodings | Supports languages, such as Chinese, Japanese, and Korean, that contain more than 256 characters. In a DBCS, a pair of code points (a double byte) represents each character. The Encoding.IsSingleByte property returns <code>false</code> for DBCS encodings. You can retrieve an encoding object for a particular DBCS by calling the Encoding.GetEncoding(Int32) method. When an application handles DBCS data, the first byte of a DBCS character (the lead byte) is processed in combination with the trail byte that immediately follows it. Because a single pair of double-byte code points can represent different characters depending on the code page, this scheme still does not allow for the combination of two languages, such as Japanese and Chinese, in the same data stream. |

These encodings enable you to work with Unicode characters as well as with encodings that are most commonly used in legacy applications. In addition, you can create a custom encoding by defining a class that derives from [Encoding](#) and overriding its members.

.NET Core encoding support

By default, .NET Core does not make available any code page encodings other than code page 28591 and the Unicode encodings, such as UTF-8 and UTF-16. However, you can add the code page encodings found in standard Windows apps that target .NET to your app. For more information, see the [CodePagesEncodingProvider](#) topic.

Selecting an Encoding Class

If you have the opportunity to choose the encoding to be used by your application, you should use a Unicode encoding, preferably either [UTF8Encoding](#) or [UnicodeEncoding](#). (.NET also supports a third Unicode encoding, [UTF32Encoding](#).)

If you are planning to use an ASCII encoding ([ASCIIEncoding](#)), choose [UTF8Encoding](#) instead. The two encodings are identical for the ASCII character set, but [UTF8Encoding](#) has the following advantages:

- It can represent every Unicode character, whereas [ASCIIEncoding](#) supports only the Unicode character values between U+0000 and U+007F.
- It provides error detection and better security.
- It has been tuned to be as fast as possible and should be faster than any other encoding. Even for content that is entirely ASCII, operations performed with [UTF8Encoding](#) are faster than operations performed with [ASCIIEncoding](#).

You should consider using [ASCIIEncoding](#) only for legacy applications. However, even for legacy applications, [UTF8Encoding](#) might be a better choice for the following reasons (assuming default settings):

- If your application has content that is not strictly ASCII and encodes it with [ASCIIEncoding](#), each non-ASCII character encodes as a question mark (?). If the application then decodes this data, the information is lost.
- If your application has content that is not strictly ASCII and encodes it with [UTF8Encoding](#), the result seems unintelligible if interpreted as ASCII. However, if the application then uses a UTF-8 decoder to decode this data, the data performs a round trip successfully.

In a web application, characters sent to the client in response to a web request should reflect the encoding used on the client. In most cases, you should set the [HttpResponse.ContentEncoding](#) property to the value returned by the [HttpRequest.ContentEncoding](#) property to display text in the encoding that the user expects.

Using an Encoding Object

An encoder converts a string of characters (most commonly, Unicode characters) to its numeric (byte) equivalent. For example, you might use an ASCII encoder to convert Unicode characters to ASCII so that they can be displayed at the console. To perform the conversion, you call the [Encoding.GetBytes](#) method. If you want to determine how many bytes are needed to store the encoded characters before performing the encoding, you can call the [GetByteCount](#) method.

The following example uses a single byte array to encode strings in two separate operations. It maintains an index that indicates the starting position in the byte array for the next set of ASCII-encoded bytes. It calls the [ASCIIEncoding.GetByteCount\(String\)](#) method to ensure that the byte array is large enough to accommodate the encoded string. It then calls the [ASCIIEncoding.GetBytes\(String, Int32, Int32, Byte\[\], Int32\)](#) method to encode the characters in the string.

C#

```
using System;
using System.Text;

public class Example
{
    public static void Main()
    {
        string[] strings= { "This is the first sentence. ",
                           "This is the second sentence. " };
        Encoding asciiEncoding = Encoding.ASCII;

        // Create array of adequate size.
        byte[] bytes = new byte[49];
        // Create index for current position of array.
        int index = 0;

        Console.WriteLine("Strings to encode:");
        foreach (var stringValue in strings) {
            Console.WriteLine("  {0}", stringValue);

            int count = asciiEncoding.GetByteCount(stringValue);
            if (count + index >= bytes.Length)
                Array.Resize(ref bytes, bytes.Length + 50);

            int written = asciiEncoding.GetBytes(stringValue, 0,
                                                 stringValue.Length,
                                                 bytes, index);

            index = index + written;
        }
        Console.WriteLine("\nEncoded bytes:");
    }
}
```

```

        Console.WriteLine("{0}", ShowByteValues(bytes, index));
        Console.WriteLine();

        // Decode Unicode byte array to a string.
        string newString = asciiEncoding.GetString(bytes, 0, index);
        Console.WriteLine("Decoded: {0}", newString);
    }

    private static string ShowByteValues(byte[] bytes, int last )
    {
        string returnString = "    ";
        for (int ctr = 0; ctr <= last - 1; ctr++) {
            if (ctr % 20 == 0)
                returnString += "\n    ";
            returnString += String.Format("{0:X2} ", bytes[ctr]);
        }
        return returnString;
    }
}

// The example displays the following output:
//      Strings to encode:
//          This is the first sentence.
//          This is the second sentence.
//
//      Encoded bytes:
//
//          54 68 69 73 20 69 73 20 74 68 65 20 66 69 72 73 74 20 73 65
//          6E 74 65 6E 63 65 2E 20 54 68 69 73 20 69 73 20 74 68 65 20
//          73 65 63 6F 6E 64 20 73 65 6E 74 65 6E 63 65 2E 20
//
//      Decoded: This is the first sentence. This is the second sentence.

```

A decoder converts a byte array that reflects a particular character encoding into a set of characters, either in a character array or in a string. To decode a byte array into a character array, you call the [Encoding.GetChars](#) method. To decode a byte array into a string, you call the [GetString](#) method. If you want to determine how many characters are needed to store the decoded bytes before performing the decoding, you can call the [GetCharCount](#) method.

The following example encodes three strings and then decodes them into a single array of characters. It maintains an index that indicates the starting position in the character array for the next set of decoded characters. It calls the [GetCharCount](#) method to ensure that the character array is large enough to accommodate all the decoded characters. It then calls the [ASCIIEncoding.GetChars\(Byte\[\], Int32, Int32, Char\[\], Int32\)](#) method to decode the byte array.

C#

```

using System;
using System.Text;

```

```
public class Example
{
    public static void Main()
    {
        string[] strings = { "This is the first sentence. ",
                            "This is the second sentence. ",
                            "This is the third sentence. " };
        Encoding asciiEncoding = Encoding.ASCII;
        // Array to hold encoded bytes.
        byte[] bytes;
        // Array to hold decoded characters.
        char[] chars = new char[50];
        // Create index for current position of character array.
        int index = 0;

        foreach (var stringValue in strings) {
            Console.WriteLine("String to Encode: {0}", stringValue);
            // Encode the string to a byte array.
            bytes = asciiEncoding.GetBytes(stringValue);
            // Display the encoded bytes.
            Console.Write("Encoded bytes: ");
            for (int ctr = 0; ctr < bytes.Length; ctr++)
                Console.Write(" {0}{1:X2}",
                             ctr % 20 == 0 ? Environment.NewLine : "",
                             bytes[ctr]);
            Console.WriteLine();

            // Decode the bytes to a single character array.
            int count = asciiEncoding.GetCharCount(bytes);
            if (count + index >= chars.Length)
                Array.Resize(ref chars, chars.Length + 50);

            int written = asciiEncoding.GetChars(bytes, 0,
                                                 bytes.Length,
                                                 chars, index);
            index = index + written;
            Console.WriteLine();
        }

        // Instantiate a single string containing the characters.
        string decodedString = new string(chars, 0, index - 1);
        Console.WriteLine("Decoded string: ");
        Console.WriteLine(decodedString);
    }
}

// The example displays the following output:
// String to Encode: This is the first sentence.
// Encoded bytes:
// 54 68 69 73 20 69 73 20 74 68 65 20 66 69 72 73 74 20 73 65
// 6E 74 65 6E 63 65 2E 20
//
// String to Encode: This is the second sentence.
// Encoded bytes:
// 54 68 69 73 20 69 73 20 74 68 65 20 73 65 63 6F 6E 64 20 73
```

```
// 65 6E 74 65 6E 63 65 2E 20
//
// String to Encode: This is the third sentence.
// Encoded bytes:
// 54 68 69 73 20 69 73 20 74 68 65 20 74 68 69 72 64 20 73 65
// 6E 74 65 6E 63 65 2E 20
//
// Decoded string:
// This is the first sentence. This is the second sentence. This is the
third sentence.
```

The encoding and decoding methods of a class derived from [Encoding](#) are designed to work on a complete set of data; that is, all the data to be encoded or decoded is supplied in a single method call. However, in some cases, data is available in a stream, and the data to be encoded or decoded may be available only from separate read operations. This requires the encoding or decoding operation to remember any saved state from its previous invocation. Methods of classes derived from [Encoder](#) and [Decoder](#) are able to handle encoding and decoding operations that span multiple method calls.

An [Encoder](#) object for a particular encoding is available from that encoding's [Encoding.GetEncoder](#) property. A [Decoder](#) object for a particular encoding is available from that encoding's [Encoding.GetDecoder](#) property. For decoding operations, note that classes derived from [Decoder](#) include a [Decoder.GetChars](#) method, but they do not have a method that corresponds to [Encoding.GetString](#).

The following example illustrates the difference between using the [Encoding.GetString](#) and [Decoder.GetChars](#) methods for decoding a Unicode byte array. The example encodes a string that contains some Unicode characters to a file, and then uses the two decoding methods to decode them ten bytes at a time. Because a surrogate pair occurs in the tenth and eleventh bytes, it is decoded in separate method calls. As the output shows, the [Encoding.GetString](#) method is not able to correctly decode the bytes and instead replaces them with U+FFF (REPLACEMENT CHARACTER). On the other hand, the [Decoder.GetChars](#) method is able to successfully decode the byte array to get the original string.

C#

```
using System;
using System.IO;
using System.Text;

public class Example
{
    public static void Main()
    {
```

```

// Use default replacement fallback for invalid encoding.
UnicodeEncoding enc = new UnicodeEncoding(true, false, false);

// Define a string with various Unicode characters.
string str1 = "AB YZ 19 \uD800\udc05 \u00e4";
str1 += "Unicode characters. \u00a9 \u010C s \u0062\u0308";
Console.WriteLine("Created original string...\n");

// Convert string to byte array.
byte[] bytes = enc.GetBytes(str1);

FileStream fs = File.Create(@".\characters.bin");
BinaryWriter bw = new BinaryWriter(fs);
bw.Write(bytes);
bw.Close();

// Read bytes from file.
FileStream fsIn = File.OpenRead(@".\characters.bin");
BinaryReader br = new BinaryReader(fsIn);

const int count = 10;           // Number of bytes to read at a time.
byte[] bytesRead = new byte[10]; // Buffer (byte array).
int read;                     // Number of bytes actually read.
string str2 = String.Empty;    // Decoded string.

// Try using Encoding object for all operations.
do {
    read = br.Read(bytesRead, 0, count);
    str2 += enc.GetString(bytesRead, 0, read);
} while (read == count);
br.Close();
Console.WriteLine("Decoded string using
UnicodeEncoding.GetString()...\"");
CompareForEquality(str1, str2);
Console.WriteLine();

// Use Decoder for all operations.
fsIn = File.OpenRead(@".\characters.bin");
br = new BinaryReader(fsIn);
Decoder decoder = enc.GetDecoder();
char[] chars = new char[50];
int index = 0;                 // Next character to write in array.
int written = 0;               // Number of chars written to array.
do {
    read = br.Read(bytesRead, 0, count);
    if (index + decoder.GetCharCount(bytesRead, 0, read) - 1 >=
chars.Length)
        Array.Resize(ref chars, chars.Length + 50);

    written = decoder.GetChars(bytesRead, 0, read, chars, index);
    index += written;
} while (read == count);
br.Close();
// Instantiate a string with the decoded characters.
string str3 = new String(chars, 0, index);

```

```

        Console.WriteLine("Decoded string using
UnicodeEncoding.Decoder.GetString()...");
        CompareForEquality(str1, str3);
    }

private static void CompareForEquality(string original, string decoded)
{
    bool result = original.Equals(decoded);
    Console.WriteLine("original = decoded: {0}",
                      original.Equals(decoded, StringComparison.OrdinalIgnoreCase));
    if (!result) {
        Console.WriteLine("Code points in original string:");
        foreach (var ch in original)
            Console.Write("{0} ", Convert.ToInt16(ch).ToString("X4"));
        Console.WriteLine();

        Console.WriteLine("Code points in decoded string:");
        foreach (var ch in decoded)
            Console.Write("{0} ", Convert.ToInt16(ch).ToString("X4"));
        Console.WriteLine();
    }
}

// The example displays the following output:
//   Created original string...
//
//   Decoded string using UnicodeEncoding.GetString()...
//   original = decoded: False
//   Code points in original string:
//   0041 0042 0020 0059 005A 0020 0031 0039 0020 D800 DC05 0020 00E4 0055
006E 0069 0063 006F
//   0064 0065 0020 0063 0068 0061 0072 0061 0063 0074 0065 0072 0073 002E
0020 00A9 0020 010C
//   0020 0073 0020 0062 0308
//   Code points in decoded string:
//   0041 0042 0020 0059 005A 0020 0031 0039 0020 FFFD FFFD 0020 00E4 0055
006E 0069 0063 006F
//   0064 0065 0020 0063 0068 0061 0072 0061 0063 0074 0065 0072 0073 002E
0020 00A9 0020 010C
//   0020 0073 0020 0062 0308
//
//   Decoded string using UnicodeEncoding.Decoder.GetString()...
//   original = decoded: True

```

Choosing a Fallback Strategy

When a method tries to encode or decode a character but no mapping exists, it must implement a fallback strategy that determines how the failed mapping should be handled. There are three types of fallback strategies:

- Best-fit fallback

- Replacement fallback
- Exception fallback

ⓘ Important

The most common problems in encoding operations occur when a Unicode character cannot be mapped to a particular code page encoding. The most common problems in decoding operations occur when invalid byte sequences cannot be translated into valid Unicode characters. For these reasons, you should know which fallback strategy a particular encoding object uses. Whenever possible, you should specify the fallback strategy used by an encoding object when you instantiate the object.

Best-Fit Fallback

When a character does not have an exact match in the target encoding, the encoder can try to map it to a similar character. (Best-fit fallback is mostly an encoding rather than a decoding issue. There are very few code pages that contain characters that cannot be successfully mapped to Unicode.) Best-fit fallback is the default for code page and double-byte character set encodings that are retrieved by the [Encoding.GetEncoding\(Int32\)](#) and [Encoding.GetEncoding\(String\)](#) overloads.

ⓘ Note

In theory, the Unicode encoding classes provided in .NET ([UTF8Encoding](#), [UnicodeEncoding](#), and [UTF32Encoding](#)) support every character in every character set, so they can be used to eliminate best-fit fallback issues.

Best-fit strategies vary for different code pages. For example, for some code pages, full-width Latin characters map to the more common half-width Latin characters. For other code pages, this mapping is not made. Even under an aggressive best-fit strategy, there is no imaginable fit for some characters in some encodings. For example, a Chinese ideograph has no reasonable mapping to code page 1252. In this case, a replacement string is used. By default, this string is just a single QUESTION MARK (U+003F).

ⓘ Note

Best-fit strategies are not documented in detail. However, several code pages are documented at the [Unicode Consortium's](#) website. Please review the [readme.txt](#)

file in that folder for a description of how to interpret the mapping files.

The following example uses code page 1252 (the Windows code page for Western European languages) to illustrate best-fit mapping and its drawbacks. The [Encoding.GetEncoding\(Int32\)](#) method is used to retrieve an encoding object for code page 1252. By default, it uses a best-fit mapping for Unicode characters that it does not support. The example instantiates a string that contains three non-ASCII characters - CIRCLED LATIN CAPITAL LETTER S (U+24C8), SUPERSCRIPT FIVE (U+2075), and INFINITY (U+221E) - separated by spaces. As the output from the example shows, when the string is encoded, the three original non-space characters are replaced by QUESTION MARK (U+003F), DIGIT FIVE (U+0035), and DIGIT EIGHT (U+0038). DIGIT EIGHT is a particularly poor replacement for the unsupported INFINITY character, and QUESTION MARK indicates that no mapping was available for the original character.

C#

```
using System;
using System.Text;

public class Example
{
    public static void Main()
    {
        // Get an encoding for code page 1252 (Western Europe character set).
        Encoding cp1252 = Encoding.GetEncoding(1252);

        // Define and display a string.
        string str = "\u24c8 \u2075 \u221e";
        Console.WriteLine("Original string: " + str);
        Console.Write("Code points in string: ");
        foreach (var ch in str)
            Console.Write("{0} ", Convert.ToInt16(ch).ToString("X4"));

        Console.WriteLine("\n");

        // Encode a Unicode string.
        Byte[] bytes = cp1252.GetBytes(str);
        Console.Write("Encoded bytes: ");
        foreach (byte byt in bytes)
            Console.Write("{0:X2} ", byt);
        Console.WriteLine("\n");

        // Decode the string.
        string str2 = cp1252.GetString(bytes);
        Console.WriteLine("String round-tripped: {0}", str.Equals(str2));
        if (! str.Equals(str2)) {
            Console.WriteLine(str2);
            foreach (var ch in str2)
                Console.Write("{0} ", Convert.ToInt16(ch).ToString("X4"));
        }
    }
}
```

```
        }
    }

// The example displays the following output:
//     Original string: ® ² °
//     Code points in string: 24C8 0020 2075 0020 221E
//
//     Encoded bytes: 3F 20 35 20 38
//
//     String round-tripped: False
//     ? 5 8
//     003F 0020 0035 0020 0038
```

Best-fit mapping is the default behavior for an [Encoding](#) object that encodes Unicode data into code page data, and there are legacy applications that rely on this behavior. However, most new applications should avoid best-fit behavior for security reasons. For example, applications should not put a domain name through a best-fit encoding.

Note

You can also implement a custom best-fit fallback mapping for an encoding. For more information, see the [Implementing a Custom Fallback Strategy](#) section.

If best-fit fallback is the default for an encoding object, you can choose another fallback strategy when you retrieve an [Encoding](#) object by calling the [Encoding.GetEncoding\(Int32, EncoderFallback, DecoderFallback\)](#) or [Encoding.GetEncoding\(String, EncoderFallback, DecoderFallback\)](#) overload. The following section includes an example that replaces each character that cannot be mapped to code page 1252 with an asterisk (*).

C#

```
using System;
using System.Text;

public class Example
{
    public static void Main()
    {
        Encoding cp1252r = Encoding.GetEncoding(1252,
            new EncoderReplacementFallback("*"),
            new DecoderReplacementFallback("*"));

        string str1 = "\u24C8 \u2075 \u221E";
        Console.WriteLine(str1);
        foreach (var ch in str1)
            Console.Write("{0} ", Convert.ToInt16(ch).ToString("X4"));
```

```

        Console.WriteLine();

        byte[] bytes = cp1252r.GetBytes(str1);
        string str2 = cp1252r.GetString(bytes);
        Console.WriteLine("Round-trip: {0}", str1.Equals(str2));
        if (! str1.Equals(str2)) {
            Console.WriteLine(str2);
            foreach (var ch in str2)
                Console.Write("{0} ", Convert.ToInt16(ch).ToString("X4"));

            Console.WriteLine();
        }
    }
}

// The example displays the following output:
//      Ⓟ ॐ
//      24C8 0020 2075 0020 221E
//      Round-trip: False
//      * * *
//      002A 0020 002A 0020 002A

```

Replacement Fallback

When a character does not have an exact match in the target scheme, but there is no appropriate character that it can be mapped to, the application can specify a replacement character or string. This is the default behavior for the Unicode decoder, which replaces any two-byte sequence that it cannot decode with REPLACEMENT_CHARACTER (U+FFFD). It is also the default behavior of the [ASCIIEncoding](#) class, which replaces each character that it cannot encode or decode with a question mark. The following example illustrates character replacement for the Unicode string from the previous example. As the output shows, each character that cannot be decoded into an ASCII byte value is replaced by 0x3F, which is the ASCII code for a question mark.

C#

```

using System;
using System.Text;

public class Example
{
    public static void Main()
    {
        Encoding enc = Encoding.ASCII;

        string str1 = "\u24C8 \u2075 \u221E";
        Console.WriteLine(str1);
        foreach (var ch in str1)
            Console.Write("{0} ", Convert.ToInt16(ch).ToString("X4"));
    }
}

```

```
Console.WriteLine("\n");

// Encode the original string using the ASCII encoder.
byte[] bytes = enc.GetBytes(str1);
Console.Write("Encoded bytes: ");
foreach (var byt in bytes)
    Console.Write("{0:X2} ", byt);
Console.WriteLine("\n");

// Decode the ASCII bytes.
string str2 = enc.GetString(bytes);
Console.WriteLine("Round-trip: {0}", str1.Equals(str2));
if (! str1.Equals(str2)) {
    Console.WriteLine(str2);
    foreach (var ch in str2)
        Console.Write("{0} ", Convert.ToInt16(ch).ToString("X4"));

    Console.WriteLine();
}
}

}

// The example displays the following output:
//      ® ¤
//      24C8 0020 2075 0020 221E
//
//      Encoded bytes: 3F 20 3F 20 3F
//
//      Round-trip: False
//      ? ? ?
//
//      003F 0020 003F 0020 003F
```

.NET includes the [EncoderReplacementFallback](#) and [DecoderReplacementFallback](#) classes, which substitute a replacement string if a character does not map exactly in an encoding or decoding operation. By default, this replacement string is a question mark, but you can call a class constructor overload to choose a different string. Typically, the replacement string is a single character, although this is not a requirement. The following example changes the behavior of the code page 1252 encoder by instantiating an [EncoderReplacementFallback](#) object that uses an asterisk (*) as a replacement string.

C#

```
using System;
using System.Text;

public class Example
{
    public static void Main()
    {
        Encoding cp1252r = Encoding.GetEncoding(1252,
                                                new EncoderReplacementFallback("*"));
    }
}
```

```

        new DecoderReplacementFallback("*"));

    string str1 = "\u24C8 \u2075 \u221E";
    Console.WriteLine(str1);
    foreach (var ch in str1)
        Console.Write("{0} ", Convert.ToInt16(ch).ToString("X4"));

    Console.WriteLine();

    byte[] bytes = cp1252r.GetBytes(str1);
    string str2 = cp1252r.GetString(bytes);
    Console.WriteLine("Round-trip: {0}", str1.Equals(str2));
    if (!str1.Equals(str2)) {
        Console.WriteLine(str2);
        foreach (var ch in str2)
            Console.Write("{0} ", Convert.ToInt16(ch).ToString("X4"));

        Console.WriteLine();
    }
}
}

// The example displays the following output:
//      Ⓜ ጀ
//      24C8 0020 2075 0020 221E
//      Round-trip: False
//      * * *
//      002A 0020 002A 0020 002A

```

Note

You can also implement a replacement class for an encoding. For more information, see the [Implementing a Custom Fallback Strategy](#) section.

In addition to QUESTION MARK (U+003F), the Unicode REPLACEMENT CHARACTER (U+FFFD) is commonly used as a replacement string, particularly when decoding byte sequences that cannot be successfully translated into Unicode characters. However, you are free to choose any replacement string, and it can contain multiple characters.

Exception Fallback

Instead of providing a best-fit fallback or a replacement string, an encoder can throw an [EncoderFallbackException](#) if it is unable to encode a set of characters, and a decoder can throw a [DecoderFallbackException](#) if it is unable to decode a byte array. To throw an exception in encoding and decoding operations, you supply an [EncoderExceptionFallback](#) object and a [DecoderExceptionFallback](#) object, respectively, to

the `Encoding.GetEncoding(String, EncoderFallback, DecoderFallback)` method. The following example illustrates exception fallback with the `ASCIIEncoding` class.

C#

```
using System;
using System.Text;

public class Example
{
    public static void Main()
    {
        Encoding enc = Encoding.GetEncoding("us-ascii",
                                            new EncoderExceptionFallback(),
                                            new DecoderExceptionFallback());

        string str1 = "\u24C8 \u2075 \u221E";
        Console.WriteLine(str1);
        foreach (var ch in str1)
            Console.Write("{0} ", Convert.ToInt16(ch).ToString("X4"));

        Console.WriteLine("\n");

        // Encode the original string using the ASCII encoder.
        byte[] bytes = {};
        try {
            bytes = enc.GetBytes(str1);
            Console.Write("Encoded bytes: ");
            foreach (var byt in bytes)
                Console.Write("{0:X2} ", byt);

            Console.WriteLine();
        }
        catch (EncoderFallbackException e) {
            Console.Write("Exception: ");
            if (e.IsUnknownSurrogate())
                Console.WriteLine("Unable to encode surrogate pair 0x{0:X4} "
                    + "0x{1:X3} at index {2}.",
                    Convert.ToInt16(e.CharUnknownHigh),
                    Convert.ToInt16(e.CharUnknownLow),
                    e.Index);
            else
                Console.WriteLine("Unable to encode 0x{0:X4} at index {1}.",
                    Convert.ToInt16(e.CharUnknown),
                    e.Index);
            return;
        }
        Console.WriteLine();

        // Decode the ASCII bytes.
        try {
            string str2 = enc.GetString(bytes);
            Console.WriteLine("Round-trip: {0}", str1.Equals(str2));
            if (!str1.Equals(str2)) {
```

```

        Console.WriteLine(str2);
        foreach (var ch in str2)
            Console.Write("{0} ", Convert.ToInt16(ch).ToString("X4"));

        Console.WriteLine();
    }
}

catch (DecoderFallbackException e) {
    Console.Write("Unable to decode byte(s) ");
    foreach (byte unknown in e.BytesUnknown)
        Console.Write("0x{0:X2} ");

    Console.WriteLine("at index {0}", e.Index);
}
}

// The example displays the following output:
//      Ⓜ ̄
//      24C8 0020 2075 0020 221E
//
//      Exception: Unable to encode 0x24C8 at index 0.

```

ⓘ Note

You can also implement a custom exception handler for an encoding operation. For more information, see the [Implementing a Custom Fallback Strategy](#) section.

The [EncoderFallbackException](#) and [DecoderFallbackException](#) objects provide the following information about the condition that caused the exception:

- The [EncoderFallbackException](#) object includes an [IsUnknownSurrogate](#) method, which indicates whether the character or characters that cannot be encoded represent an unknown surrogate pair (in which case, the method returns `true`) or an unknown single character (in which case, the method returns `false`). The characters in the surrogate pair are available from the [EncoderFallbackException.CharUnknownHigh](#) and [EncoderFallbackException.CharUnknownLow](#) properties. The unknown single character is available from the [EncoderFallbackException.CharUnknown](#) property. The [EncoderFallbackException.Index](#) property indicates the position in the string at which the first character that could not be encoded was found.
- The [DecoderFallbackException](#) object includes a [BytesUnknown](#) property that returns an array of bytes that cannot be decoded. The [DecoderFallbackException.Index](#) property indicates the starting position of the unknown bytes.

Although the [EncoderFallbackException](#) and [DecoderFallbackException](#) objects provide adequate diagnostic information about the exception, they do not provide access to the encoding or decoding buffer. Therefore, they do not allow invalid data to be replaced or corrected within the encoding or decoding method.

Implementing a Custom Fallback Strategy

In addition to the best-fit mapping that is implemented internally by code pages, .NET includes the following classes for implementing a fallback strategy:

- Use [EncoderReplacementFallback](#) and [EncoderReplacementFallbackBuffer](#) to replace characters in encoding operations.
- Use [DecoderReplacementFallback](#) and [DecoderReplacementFallbackBuffer](#) to replace characters in decoding operations.
- Use [EncoderExceptionFallback](#) and [EncoderExceptionFallbackBuffer](#) to throw an [EncoderFallbackException](#) when a character cannot be encoded.
- Use [DecoderExceptionFallback](#) and [DecoderExceptionFallbackBuffer](#) to throw a [DecoderFallbackException](#) when a character cannot be decoded.

In addition, you can implement a custom solution that uses best-fit fallback, replacement fallback, or exception fallback, by following these steps:

1. Derive a class from [EncoderFallback](#) for encoding operations, and from [DecoderFallback](#) for decoding operations.
2. Derive a class from [EncoderFallbackBuffer](#) for encoding operations, and from [DecoderFallbackBuffer](#) for decoding operations.
3. For exception fallback, if the predefined [EncoderFallbackException](#) and [DecoderFallbackException](#) classes do not meet your needs, derive a class from an exception object such as [Exception](#) or [ArgumentException](#).

Deriving from EncoderFallback or DecoderFallback

To implement a custom fallback solution, you must create a class that inherits from [EncoderFallback](#) for encoding operations, and from [DecoderFallback](#) for decoding operations. Instances of these classes are passed to the [Encoding.GetEncoding\(String, EncoderFallback, DecoderFallback\)](#) method and serve as the intermediary between the encoding class and the fallback implementation.

When you create a custom fallback solution for an encoder or decoder, you must implement the following members:

- The [EncoderFallback.MaxCharCount](#) or [DecoderFallback.MaxCharCount](#) property, which returns the maximum possible number of characters that the best-fit, replacement, or exception fallback can return to replace a single character. For a custom exception fallback, its value is zero.
- The [EncoderFallback.CreateFallbackBuffer](#) or [DecoderFallback.CreateFallbackBuffer](#) method, which returns your custom [EncoderFallbackBuffer](#) or [DecoderFallbackBuffer](#) implementation. The method is called by the encoder when it encounters the first character that it is unable to successfully encode, or by the decoder when it encounters the first byte that it is unable to successfully decode.

Deriving from EncoderFallbackBuffer or DecoderFallbackBuffer

To implement a custom fallback solution, you must also create a class that inherits from [EncoderFallbackBuffer](#) for encoding operations, and from [DecoderFallbackBuffer](#) for decoding operations. Instances of these classes are returned by the [CreateFallbackBuffer](#) method of the [EncoderFallback](#) and [DecoderFallback](#) classes. The [EncoderFallback.CreateFallbackBuffer](#) method is called by the encoder when it encounters the first character that it is not able to encode, and the [DecoderFallback.CreateFallbackBuffer](#) method is called by the decoder when it encounters one or more bytes that it is not able to decode. The [EncoderFallbackBuffer](#) and [DecoderFallbackBuffer](#) classes provide the fallback implementation. Each instance represents a buffer that contains the fallback characters that will replace the character that cannot be encoded or the byte sequence that cannot be decoded.

When you create a custom fallback solution for an encoder or decoder, you must implement the following members:

- The [EncoderFallbackBuffer.Fallback](#) or [DecoderFallbackBuffer.Fallback](#) method. [EncoderFallbackBuffer.Fallback](#) is called by the encoder to provide the fallback buffer with information about the character that it cannot encode. Because the character to be encoded may be a surrogate pair, this method is overloaded. One overload is passed the character to be encoded and its index in the string. The second overload is passed the high and low surrogate along with its index in the string. The [DecoderFallbackBuffer.Fallback](#) method is called by the decoder to provide the fallback buffer with information about the bytes that it cannot decode. This method is passed an array of bytes that it cannot decode, along with the index of the first byte. The fallback method should return `true` if the fallback buffer can

supply a best-fit or replacement character or characters; otherwise, it should return `false`. For an exception fallback, the fallback method should throw an exception.

- The [EncoderFallbackBuffer.GetNextChar](#) or [DecoderFallbackBuffer.GetNextChar](#) method, which is called repeatedly by the encoder or decoder to get the next character from the fallback buffer. When all fallback characters have been returned, the method should return U+0000.
- The [EncoderFallbackBuffer.Remaining](#) or [DecoderFallbackBuffer.Remaining](#) property, which returns the number of characters remaining in the fallback buffer.
- The [EncoderFallbackBuffer.MovePrevious](#) or [DecoderFallbackBuffer.MovePrevious](#) method, which moves the current position in the fallback buffer to the previous character.
- The [EncoderFallbackBuffer.Reset](#) or [DecoderFallbackBuffer.Reset](#) method, which reinitializes the fallback buffer.

If the fallback implementation is a best-fit fallback or a replacement fallback, the classes derived from [EncoderFallbackBuffer](#) and [DecoderFallbackBuffer](#) also maintain two private instance fields: the exact number of characters in the buffer; and the index of the next character in the buffer to return.

An EncoderFallback Example

An earlier example used replacement fallback to replace Unicode characters that did not correspond to ASCII characters with an asterisk (*). The following example uses a custom best-fit fallback implementation instead to provide a better mapping of non-ASCII characters.

The following code defines a class named `CustomMapper` that is derived from [EncoderFallback](#) to handle the best-fit mapping of non-ASCII characters. Its `CreateFallbackBuffer` method returns a `CustomMapperFallbackBuffer` object, which provides the [EncoderFallbackBuffer](#) implementation. The `CustomMapper` class uses a `Dictionary< TKey, TValue >` object to store the mappings of unsupported Unicode characters (the key value) and their corresponding 8-bit characters (which are stored in two consecutive bytes in a 64-bit integer). To make this mapping available to the fallback buffer, the `CustomMapper` instance is passed as a parameter to the `CustomMapperFallbackBuffer` class constructor. Because the longest mapping is the string "INF" for the Unicode character U+221E, the `MaxCharCount` property returns 3.

C#

```

public class CustomMapper : EncoderFallback
{
    public string DefaultString;
    internal Dictionary<ushort, ulong> mapping;

    public CustomMapper() : this("*")
    {
    }

    public CustomMapper(string defaultString)
    {
        this.DefaultString = defaultString;

        // Create table of mappings
        mapping = new Dictionary<ushort, ulong>();
        mapping.Add(0x24C8, 0x53);
        mapping.Add(0x2075, 0x35);
        mapping.Add(0x221E, 0x49004E0046);
    }

    public override EncoderFallbackBuffer CreateFallbackBuffer()
    {
        return new CustomMapperFallbackBuffer(this);
    }

    public override int MaxCharCount
    {
        get { return 3; }
    }
}

```

The following code defines the `CustomMapperFallbackBuffer` class, which is derived from `EncoderFallbackBuffer`. The dictionary that contains best-fit mappings and that is defined in the `CustomMapper` instance is available from its class constructor. Its `Fallback` method returns `true` if any of the Unicode characters that the ASCII encoder cannot encode are defined in the mapping dictionary; otherwise, it returns `false`. For each fallback, the private `count` variable indicates the number of characters that remain to be returned, and the private `index` variable indicates the position in the string buffer, `charsToReturn`, of the next character to return.

C#

```

public class CustomMapperFallbackBuffer : EncoderFallbackBuffer
{
    int count = -1;                                // Number of characters to return
    int index = -1;                                 // Index of character to return
    CustomMapper fb;
    string charsToReturn;

```

```
public CustomMapperFallbackBuffer(CustomMapper fallback)
{
    this.fb = fallback;
}

public override bool Fallback(char charUnknownHigh, char charUnknownLow,
int index)
{
    // Do not try to map surrogates to ASCII.
    return false;
}

public override bool Fallback(char charUnknown, int index)
{
    // Return false if there are already characters to map.
    if (count >= 1) return false;

    // Determine number of characters to return.
    charsToReturn = String.Empty;

    ushort key = Convert.ToInt16(charUnknown);
    if (fb.mapping.ContainsKey(key)) {
        byte[] bytes = BitConverter.GetBytes(fb.mapping[key]);
        int ctr = 0;
        foreach (var byt in bytes) {
            if (byt > 0) {
                ctr++;
                charsToReturn += (char) byt;
            }
        }
        count = ctr;
    }
    else {
        // Return default.
        charsToReturn = fb.DefaultString;
        count = 1;
    }
    this.index = charsToReturn.Length - 1;

    return true;
}

public override char GetNextChar()
{
    // We'll return a character if possible, so subtract from the count of
    // chars to return.
    count--;
    // If count is less than zero, we've returned all characters.
    if (count < 0)
        return '\u0000';

    this.index--;
    return charsToReturn[this.index + 1];
}
```

```

public override bool MovePrevious()
{
    // Original: if count >= -1 and pos >= 0
    if (count >= -1) {
        count++;
        return true;
    }
    else {
        return false;
    }
}

public override int Remaining
{
    get { return count < 0 ? 0 : count; }
}

public override void Reset()
{
    count = -1;
    index = -1;
}
}

```

The following code then instantiates the `CustomMapper` object and passes an instance of it to the `Encoding.GetEncoding(String, EncoderFallback, DecoderFallback)` method. The output indicates that the best-fit fallback implementation successfully handles the three non-ASCII characters in the original string.

C#

```

using System;
using System.Collections.Generic;
using System.Text;

class Program
{
    static void Main()
    {
        Encoding enc = Encoding.GetEncoding("us-ascii", new CustomMapper(),
new DecoderExceptionFallback());

        string str1 = "\u24C8 \u2075 \u221E";
        Console.WriteLine(str1);
        for (int ctr = 0; ctr <= str1.Length - 1; ctr++) {
            Console.Write("{0} ", Convert.ToInt16(str1[ctr]).ToString("X4"));
            if (ctr == str1.Length - 1)
                Console.WriteLine();
        }
        Console.WriteLine();

        // Encode the original string using the ASCII encoder.
    }
}

```

```
byte[] bytes = enc.GetBytes(str1);
Console.WriteLine("Encoded bytes: ");
foreach (var byt in bytes)
    Console.WriteLine("{0:X2} ", byt);

Console.WriteLine("\n");

// Decode the ASCII bytes.
string str2 = enc.GetString(bytes);
Console.WriteLine("Round-trip: {0}", str1.Equals(str2));
if (!str1.Equals(str2)) {
    Console.WriteLine(str2);
    foreach (var ch in str2)
        Console.Write("{0} ", Convert.ToInt16(ch).ToString("X4"));

    Console.WriteLine();
}
}
```

See also

- [Introduction to character encoding in .NET](#)
- [Encoder](#)
- [Decoder](#)
- [DecoderFallback](#)
- [Encoding](#)
- [EncoderFallback](#)
- [Globalization and localization](#)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback [here](#).

 [Open a documentation issue](#)

 [Provide product feedback](#)

Best practices for comparing strings in .NET

Article • 02/01/2023

.NET provides extensive support for developing localized and globalized applications, and makes it easy to apply the conventions of either the current culture or a specific culture when performing common operations such as sorting and displaying strings. But sorting or comparing strings isn't always a culture-sensitive operation. For example, strings that are used internally by an application typically should be handled identically across all cultures. When culturally independent string data, such as XML tags, HTML tags, user names, file paths, and the names of system objects, are interpreted as if they were culture-sensitive, application code can be subject to subtle bugs, poor performance, and, in some cases, security issues.

This article examines the string sorting, comparison, and casing methods in .NET, presents recommendations for selecting an appropriate string-handling method, and provides additional information about string-handling methods.

Recommendations for string usage

When you develop with .NET, follow these recommendations when you compare strings.

Tip

Various string-related methods perform comparison. Examples include [String.Equals](#), [String.Compare](#), [String.IndexOf](#), and [String.StartsWith](#).

- Use overloads that explicitly specify the string comparison rules for string operations. Typically, this involves calling a method overload that has a parameter of type [StringComparison](#).
- Use [StringComparison.Ordinal](#) or [StringComparison.OrdinalIgnoreCase](#) for comparisons as your safe default for culture-agnostic string matching.
- Use comparisons with [StringComparison.Ordinal](#) or [StringComparison.OrdinalIgnoreCase](#) for better performance.
- Use string operations that are based on [StringComparison.CurrentCulture](#) when you display output to the user.
- Use the non-linguistic [StringComparison.Ordinal](#) or [StringComparison.OrdinalIgnoreCase](#) values instead of string operations based on

`CultureInfo.InvariantCulture` when the comparison is linguistically irrelevant (symbolic, for example).

- Use the `String.ToUpperInvariant` method instead of the `String.ToLowerInvariant` method when you normalize strings for comparison.
- Use an overload of the `String.Equals` method to test whether two strings are equal.
- Use the `String.Compare` and `String.CompareTo` methods to sort strings, not to check for equality.
- Use culture-sensitive formatting to display non-string data, such as numbers and dates, in a user interface. Use formatting with the `invariant culture` to persist non-string data in string form.

Avoid the following practices when you compare strings:

- Don't use overloads that don't explicitly or implicitly specify the string comparison rules for string operations.
- Don't use string operations based on `StringComparison.InvariantCulture` in most cases. One of the few exceptions is when you're persisting linguistically meaningful but culturally agnostic data.
- Don't use an overload of the `String.Compare` or `CompareTo` method and test for a return value of zero to determine whether two strings are equal.

Specifying string comparisons explicitly

Most of the string manipulation methods in .NET are overloaded. Typically, one or more overloads accept default settings, whereas others accept no defaults and instead define the precise way in which strings are to be compared or manipulated. Most of the methods that don't rely on defaults include a parameter of type `StringComparison`, which is an enumeration that explicitly specifies rules for string comparison by culture and case. The following table describes the `StringComparison` enumeration members.

| StringComparison member | Description |
|---|---|
| <code>.CurrentCulture</code> | Performs a case-sensitive comparison using the current culture. |
| <code>.CurrentCultureIgnoreCase</code> | Performs a case-insensitive comparison using the current culture. |
| <code>InvariantCulture</code> | Performs a case-sensitive comparison using the invariant culture. |
| <code>InvariantCultureIgnoreCase</code> | Performs a case-insensitive comparison using the invariant culture. |
| <code>Ordinal</code> | Performs an ordinal comparison. |
| <code>OrdinalIgnoreCase</code> | Performs a case-insensitive ordinal comparison. |

For example, the [IndexOf](#) method, which returns the index of a substring in a [String](#) object that matches either a character or a string, has nine overloads:

- [IndexOf\(Char\)](#), [IndexOf\(Char, Int32\)](#), and [IndexOf\(Char, Int32, Int32\)](#), which by default performs an ordinal (case-sensitive and culture-insensitive) search for a character in the string.
- [IndexOf\(String\)](#), [IndexOf\(String, Int32\)](#), and [IndexOf\(String, Int32, Int32\)](#), which by default performs a case-sensitive and culture-sensitive search for a substring in the string.
- [IndexOf\(String, StringComparison\)](#), [IndexOf\(String, Int32, StringComparison\)](#), and [IndexOf\(String, Int32, Int32, StringComparison\)](#), which include a parameter of type [StringComparison](#) that allows the form of the comparison to be specified.

We recommend that you select an overload that doesn't use default values, for the following reasons:

- Some overloads with default parameters (those that search for a [Char](#) in the string instance) perform an ordinal comparison, whereas others (those that search for a string in the string instance) are culture-sensitive. It's difficult to remember which method uses which default value, and easy to confuse the overloads.
- The intent of the code that relies on default values for method calls isn't clear. In the following example, which relies on defaults, it's difficult to know whether the developer actually intended an ordinal or a linguistic comparison of two strings, or whether a case difference between `url.Scheme` and "https" might cause the test for equality to return `false`.

C#

```
Uri url = new("https://learn.microsoft.com/");

// Incorrect
if (string.Equals(url.Scheme, "https"))
{
    // ...Code to handle HTTPS protocol.
}
```

In general, we recommend that you call a method that doesn't rely on defaults, because it makes the intent of the code unambiguous. This, in turn, makes the code more readable and easier to debug and maintain. The following example addresses the questions raised about the previous example. It makes it clear that ordinal comparison is used and that differences in case are ignored.

C#

```
Uri url = new("https://learn.microsoft.com/");

// Correct
if (string.Equals(url.Scheme, "https", StringComparison.OrdinalIgnoreCase))
{
    // ...Code to handle HTTPS protocol.
}
```

The details of string comparison

String comparison is the heart of many string-related operations, particularly sorting and testing for equality. Strings sort in a determined order: If "my" appears before "string" in a sorted list of strings, "my" must compare less than or equal to "string". Additionally, comparison implicitly defines equality. The comparison operation returns zero for strings it deems equal. A good interpretation is that neither string is less than the other. Most meaningful operations involving strings include one or both of these procedures: comparing with another string, and executing a well-defined sort operation.

Note

You can download the [Sorting Weight Tables](#), a set of text files that contain information on the character weights used in sorting and comparison operations for Windows operating systems, and the [Default Unicode Collation Element Table](#), the latest version of the sort weight table for Linux and macOS. The specific version of the sort weight table on Linux and macOS depends on the version of the [International Components for Unicode](#) libraries installed on the system. For information on ICU versions and the Unicode versions that they implement, see [Downloading ICU](#).

However, evaluating two strings for equality or sort order doesn't yield a single, correct result; the outcome depends on the criteria used to compare the strings. In particular, string comparisons that are ordinal or that are based on the casing and sorting conventions of the current culture or the [invariant culture](#) (a locale-agnostic culture based on the English language) may produce different results.

In addition, string comparisons using different versions of .NET or using .NET on different operating systems or operating system versions may return different results. For more information, see [Strings and the Unicode Standard](#).

String comparisons that use the current culture

One criterion involves using the conventions of the current culture when comparing strings. Comparisons that are based on the current culture use the thread's current culture or locale. If the culture isn't set by the user, it defaults to the operating system's setting. You should always use comparisons that are based on the current culture when data is linguistically relevant, and when it reflects culture-sensitive user interaction.

However, comparison and casing behavior in .NET changes when the culture changes. This happens when an application executes on a computer that has a different culture than the computer on which the application was developed, or when the executing thread changes its culture. This behavior is intentional, but it remains non-obvious to many developers. The following example illustrates differences in sort order between the U.S. English ("en-US") and Swedish ("sv-SE") cultures. Note that the words "ångström", "Windows", and "Visual Studio" appear in different positions in the sorted string arrays.

C#

```
using System.Globalization;

// Words to sort
string[] values= { "able", "ångström", "apple", "Æble",
                   "Windows", "Visual Studio" };

// Current culture
Array.Sort(values);
DisplayArray(values);

// Change culture to Swedish (Sweden)
string originalCulture = CultureInfo.CurrentCulture.Name;
Thread.CurrentThread.CurrentCulture = new CultureInfo("sv-SE");
Array.Sort(values);
DisplayArray(values);

// Restore the original culture
Thread.CurrentThread.CurrentCulture = new CultureInfo(originalCulture);

static void DisplayArray(string[] values)
{
    Console.WriteLine($"Sorting using the {CultureInfo.CurrentCulture.Name}
culture:");

    foreach (string value in values)
        Console.WriteLine($"{value}");

    Console.WriteLine();
}

// The example displays the following output:
//      Sorting using the en-US culture:
//          able
```

```
//      Äble
//      ångström
//      apple
//      Visual Studio
//      Windows
//
//      Sorting using the sv-SE culture:
//      able
//      apple
//      Visual Studio
//      Windows
//      ångström
//      Äble
```

Case-insensitive comparisons that use the current culture are the same as culture-sensitive comparisons, except that they ignore case as dictated by the thread's current culture. This behavior may manifest itself in sort orders as well.

Comparisons that use current culture semantics are the default for the following methods:

- [String.Compare](#) overloads that don't include a [StringComparison](#) parameter.
- [String.CompareTo](#) overloads.
- The default [String.StartsWith\(String\)](#) method, and the [String.StartsWith\(String, Boolean, CultureInfo\)](#) method with a `null` [CultureInfo](#) parameter.
- The default [String.EndsWith\(String\)](#) method, and the [String.EndsWith\(String, Boolean, CultureInfo\)](#) method with a `null` [CultureInfo](#) parameter.
- [String.IndexOf](#) overloads that accept a [String](#) as a search parameter and that don't have a [StringComparison](#) parameter.
- [String.LastIndexOf](#) overloads that accept a [String](#) as a search parameter and that don't have a [StringComparison](#) parameter.

In any case, we recommend that you call an overload that has a [StringComparison](#) parameter to make the intent of the method call clear.

Subtle and not so subtle bugs can emerge when non-linguistic string data is interpreted linguistically, or when string data from a particular culture is interpreted using the conventions of another culture. The canonical example is the Turkish-I problem.

For nearly all Latin alphabets, including U.S. English, the character "i" (\u0069) is the lowercase version of the character "I" (\u0049). This casing rule quickly becomes the default for someone programming in such a culture. However, the Turkish ("tr-TR") alphabet includes an "I with a dot" character "İ" (\u0130), which is the capital version of "i". Turkish also includes a lowercase "i without a dot" character, "ı" (\u0131), which capitalizes to "I". This behavior occurs in the Azerbaijani ("az") culture as well.

Therefore, assumptions made about capitalizing "i" or lowercasing "I" aren't valid among all cultures. If you use the default overloads for string comparison routines, they will be subject to variance between cultures. If the data to be compared is non-linguistic, using the default overloads can produce undesirable results, as the following attempt to perform a case-insensitive comparison of the strings "bill" and "BILL" illustrates.

C#

```
using System.Globalization;

string name = "Bill";

Thread.CurrentThread.CurrentCulture = new CultureInfo("en-US");
Console.WriteLine($"Culture =
{Thread.CurrentThread.CurrentCulture.DisplayName}");
Console.WriteLine($"    Is 'Bill' the same as 'BILL'? {name.Equals("BILL",
 StringComparison.OrdinalIgnoreCase)}");
Console.WriteLine($"    Does 'Bill' start with 'BILL'?
{name.StartsWith("BILL", true, null)}");
Console.WriteLine();

Thread.CurrentThread.CurrentCulture = new CultureInfo("tr-TR");
Console.WriteLine($"Culture =
{Thread.CurrentThread.CurrentCulture.DisplayName}");
Console.WriteLine($"    Is 'Bill' the same as 'BILL'? {name.Equals("BILL",
 StringComparison.OrdinalIgnoreCase)}");
Console.WriteLine($"    Does 'Bill' start with 'BILL'?
{name.StartsWith("BILL", true, null)}");

// The example displays the following output:
//'
//'
//    Culture = English (United States)
//        Is 'Bill' the same as 'BILL'? True
//        Does 'Bill' start with 'BILL'? True
//'
//    Culture = Turkish (Türkçe)
//        Is 'Bill' the same as 'BILL'? True
//        Does 'Bill' start with 'BILL'? False
```

This comparison could cause significant problems if the culture is inadvertently used in security-sensitive settings, as in the following example. A method call such as `IsFileURI("file:")` returns `true` if the current culture is U.S. English, but `false` if the current culture is Turkish. Thus, on Turkish systems, someone could circumvent security measures that block access to case-insensitive URIs that begin with "FILE:".

C#

```
public static bool IsFileURI(string path) =>
    path.StartsWith("FILE:", true, null);
```

In this case, because "file:" is meant to be interpreted as a non-linguistic, culture-insensitive identifier, the code should instead be written as shown in the following example:

C#

```
public static bool IsFileURI(string path) =>
    path.StartsWith("FILE:", StringComparison.OrdinalIgnoreCase);
```

Ordinal string operations

Specifying the `StringComparison.Ordinal` or `StringComparison.OrdinalIgnoreCase` value in a method call signifies a non-linguistic comparison in which the features of natural languages are ignored. Methods that are invoked with these `StringComparison` values base string operation decisions on simple byte comparisons instead of casing or equivalence tables that are parameterized by culture. In most cases, this approach best fits the intended interpretation of strings while making code faster and more reliable.

Ordinal comparisons are string comparisons in which each byte of each string is compared without linguistic interpretation; for example, "windows" doesn't match "Windows". This is essentially a call to the C runtime `strcmp` function. Use this comparison when the context dictates that strings should be matched exactly or demands conservative matching policy. Additionally, ordinal comparison is the fastest comparison operation because it applies no linguistic rules when determining a result.

Strings in .NET can contain embedded null characters (and other non-printing characters). One of the clearest differences between ordinal and culture-sensitive comparison (including comparisons that use the invariant culture) concerns the handling of embedded null characters in a string. These characters are ignored when you use the `String.Compare` and `String.Equals` methods to perform culture-sensitive comparisons (including comparisons that use the invariant culture). As a result, strings that contain embedded null characters can be considered equal to strings that don't. Embedded non-printing characters might be skipped for the purpose of string comparison methods, such as `String.StartsWith`.

ⓘ Important

Although string comparison methods disregard embedded null characters, string search methods such as `String.Contains`, `String.EndsWith`, `String.IndexOf`, `String.LastIndexOf`, and `String.StartsWith` do not.

The following example performs a culture-sensitive comparison of the string "Aa" with a similar string that contains several embedded null characters between "A" and "a", and shows how the two strings are considered equal:

C#

```
string str1 = "Aa";
string str2 = "A" + new string('\u0000', 3) + "a";

Thread.CurrentThread.CurrentCulture =
System.Globalization.CultureInfo.GetCultureInfo("en-us");

Console.WriteLine($"Comparing '{str1}' ({ShowBytes(str1)}) and '{str2}'"
({ShowBytes(str2)}):");
Console.WriteLine("    With String.Compare:");
Console.WriteLine($"        Current Culture: {string.Compare(str1, str2,
 StringComparison.CurrentCulture)}");
Console.WriteLine($"        Invariant Culture: {string.Compare(str1, str2,
 StringComparison.InvariantCulture)}");
Console.WriteLine("    With String.Equals:");
Console.WriteLine($"        Current Culture: {string.Equals(str1, str2,
 StringComparison.CurrentCulture)}");
Console.WriteLine($"        Invariant Culture: {string.Equals(str1, str2,
 StringComparison.InvariantCulture)}");

string ShowBytes(string value)
{
    string hexString = string.Empty;
    for (int index = 0; index < value.Length; index++)
    {
        string result = Convert.ToInt32(value[index]).ToString("X4");
        result = string.Concat(" ", result.Substring(0,2), " ",
result.Substring(2, 2));
        hexString += result;
    }
    return hexString.Trim();
}

// The example displays the following output:
// Comparing 'Aa' (00 41 00 61) and 'Aa' (00 41 00 00 00 00 00 00 00
61):
//     With String.Compare:
//         Current Culture: 0
//         Invariant Culture: 0
//     With String.Equals:
//         Current Culture: True
//         Invariant Culture: True
```

However, the strings aren't considered equal when you use ordinal comparison, as the following example shows:

C#

```
string str1 = "Aa";
string str2 = "A" + new String('\u0000', 3) + "a";

Console.WriteLine($"Comparing '{str1}' ({ShowBytes(str1)}) and '{str2}'"
({ShowBytes(str2)}):");
Console.WriteLine("    With String.Compare:");
Console.WriteLine($"        Ordinal: {string.Compare(str1, str2,
 StringComparison.OrdinalIgnoreCase)}");
Console.WriteLine("    With String.Equals:");
Console.WriteLine($"        Ordinal: {string.Equals(str1, str2,
 StringComparison.OrdinalIgnoreCase)}");

string ShowBytes(string str)
{
    string hexString = string.Empty;
    for (int ctr = 0; ctr < str.Length; ctr++)
    {
        string result = Convert.ToInt32(str[ctr]).ToString("X4");
        result = " " + result.Substring(0, 2) + " " + result.Substring(2,
2);
        hexString += result;
    }
    return hexString.Trim();
}

// The example displays the following output:
// Comparing 'Aa' (00 41 00 61) and 'A a' (00 41 00 00 00 00 00 00
61):
//     With String.Compare:
//         Ordinal: 97
//     With String.Equals:
//         Ordinal: False
```

Case-insensitive ordinal comparisons are the next most conservative approach. These comparisons ignore most casing; for example, "windows" matches "Windows". When dealing with ASCII characters, this policy is equivalent to `StringComparison.OrdinalIgnoreCase`, except that it ignores the usual ASCII casing. Therefore, any character in [A, Z] (\u0041-\u005A) matches the corresponding character in [a,z] (\u0061-\u007A). Casing outside the ASCII range uses the invariant culture's tables. Therefore, the following comparison:

C#

```
string.Compare(strA, strB, StringComparison.OrdinalIgnoreCase);
```

is equivalent to (but faster than) this comparison:

C#

```
string.Compare(strA.ToUpperInvariant(), strB.ToUpperInvariant(),
StringComparison.OrdinalIgnoreCase);
```

These comparisons are still very fast.

Both [StringComparison.Ordinal](#) and [StringComparison.OrdinalIgnoreCase](#) use the binary values directly, and are best suited for matching. When you aren't sure about your comparison settings, use one of these two values. However, because they perform a byte-by-byte comparison, they don't sort by a linguistic sort order (like an English dictionary) but by a binary sort order. The results may look odd in most contexts if displayed to users.

Ordinal semantics are the default for [String.Equals](#) overloads that don't include a [StringComparison](#) argument (including the equality operator). In any case, we recommend that you call an overload that has a [StringComparison](#) parameter.

String operations that use the invariant culture

Comparisons with the invariant culture use the [CompareInfo](#) property returned by the static [CultureInfo.InvariantCulture](#) property. This behavior is the same on all systems; it translates any characters outside its range into what it believes are equivalent invariant characters. This policy can be useful for maintaining one set of string behavior across cultures, but it often provides unexpected results.

Case-insensitive comparisons with the invariant culture use the static [CompareInfo](#) property returned by the static [CultureInfo.InvariantCulture](#) property for comparison information as well. Any case differences among these translated characters are ignored.

Comparisons that use [StringComparison.InvariantCulture](#) and [StringComparison.OrdinalIgnoreCase](#) work identically on ASCII strings. However, [StringComparison.InvariantCulture](#) makes linguistic decisions that might not be appropriate for strings that have to be interpreted as a set of bytes. The [CultureInfo.InvariantCulture.CompareInfo](#) object makes the [Compare](#) method interpret certain sets of characters as equivalent. For example, the following equivalence is valid under the invariant culture:

InvariantCulture: a + ° = å

The LATIN SMALL LETTER A character "a" (\u0061), when it's next to the COMBINING RING ABOVE character "+" "\u030a", is interpreted as the LATIN SMALL LETTER A WITH RING ABOVE character "å" (\u00e5). As the following example shows, this behavior differs from ordinal comparison.

C#

```
string separated = "\u0061\u030a";
string combined = "\u00e5";

Console.WriteLine("Equal sort weight of {0} and {1} using InvariantCulture:
{2}",
separated, combined,
string.Compare(separated, combined,
StringComparison.InvariantCulture) == 0);

Console.WriteLine("Equal sort weight of {0} and {1} using Ordinal: {2}",
separated, combined,
string.Compare(separated, combined,
StringComparison.Ordinal) == 0);

// The example displays the following output:
//      Equal sort weight of aº and å using InvariantCulture: True
//      Equal sort weight of aº and å using Ordinal: False
```

When interpreting file names, cookies, or anything else where a combination such as "å" can appear, ordinal comparisons still offer the most transparent and fitting behavior.

On balance, the invariant culture has few properties that make it useful for comparison. It does comparison in a linguistically relevant manner, which prevents it from guaranteeing full symbolic equivalence, but it isn't the choice for display in any culture. One of the few reasons to use [StringComparison.InvariantCulture](#) for comparison is to persist ordered data for a cross-culturally identical display. For example, if a large data file that contains a list of sorted identifiers for display accompanies an application, adding to this list would require an insertion with invariant-style sorting.

Choosing a StringComparison member for your method call

The following table outlines the mapping from semantic string context to a [StringComparison](#) enumeration member:

| Data | Behavior | Corresponding System.StringComparison |
|--------------------------------------|---|--|
| | | value |
| Case-sensitive internal identifiers. | A non-linguistic identifier, where bytes match exactly. | Ordinal |
| | | |

| Data | Behavior | Corresponding System.StringComparison value |
|---|---|--|
| standards such as XML and HTTP. | Case-sensitive security-related settings. | |
| Case-insensitive internal identifiers. | A non-linguistic identifier, where case is irrelevant. | OrdinalIgnoreCase |
| Case-insensitive identifiers in standards such as XML and HTTP. | | |
| File paths. | | |
| Registry keys and values. | | |
| Environment variables. | | |
| Resource identifiers (for example, handle names). | | |
| Case-insensitive security-related settings. | | |
| Some persisted, linguistically relevant data. | Culturally agnostic data that still is linguistically relevant. | InvariantCulture -or- InvariantCultureIgnoreCase |
| Display of linguistic data that requires a fixed sort order. | | |
| Data displayed to the user. | Data that requires local linguistic customs. | CurrentCulture -or- CurrentCultureIgnoreCase |
| Most user input. | | |

Common string comparison methods in .NET

The following sections describe the methods that are most commonly used for string comparison.

String.Compare

Default interpretation: [StringComparison.CurrentCulture](#).

As the operation most central to string interpretation, all instances of these method calls should be examined to determine whether strings should be interpreted according to the current culture, or dissociated from the culture (symbolically). Typically, it's the latter, and a [StringComparison.Ordinal](#) comparison should be used instead.

The [System.Globalization.CompareInfo](#) class, which is returned by the [CultureInfo.CompareInfo](#) property, also includes a [Compare](#) method that provides a large number of matching options (ordinal, ignoring white space, ignoring kana type, and so on) by means of the [CompareOptions](#) flag enumeration.

String.CompareTo

Default interpretation: [StringComparison.CurrentCulture](#).

This method doesn't currently offer an overload that specifies a [StringComparison](#) type. It's usually possible to convert this method to the recommended [String.Compare\(String, String, StringComparison\)](#) form.

Types that implement the [IComparable](#) and [IComparable<T>](#) interfaces implement this method. Because it doesn't offer the option of a [StringComparison](#) parameter, implementing types often let the user specify a [StringComparer](#) in their constructor. The following example defines a [FileName](#) class whose class constructor includes a [StringComparer](#) parameter. This [StringComparer](#) object is then used in the [FileName.CompareTo](#) method.

C#

```
class FileName : IComparable
{
    private readonly StringComparer _comparer;

    public string Name { get; }

    public FileName(string name, StringComparer? comparer)
    {
        if (string.IsNullOrEmpty(name)) throw new
ArgumentNullException(nameof(name));

        Name = name;

        if (comparer != null)
            _comparer = comparer;
    }
}
```

```

        else
            _comparer = StringComparer.OrdinalIgnoreCase;
    }

    public int CompareTo(object? obj)
    {
        if (obj == null) return 1;

        if (obj is not FileName)
            return _comparer.Compare(Name, obj.ToString());
        else
            return _comparer.Compare(Name, ((FileName)obj).Name);
    }
}

```

String.Equals

Default interpretation: [StringComparison.Ordinal](#).

The [String](#) class lets you test for equality by calling either the static or instance [Equals](#) method overloads, or by using the static equality operator. The overloads and operator use ordinal comparison by default. However, we still recommend that you call an overload that explicitly specifies the [StringComparison](#) type even if you want to perform an ordinal comparison; this makes it easier to search code for a certain string interpretation.

String.ToUpper and String.ToLower

Default interpretation: [StringComparison.CurrentCulture](#).

Be careful when you use the [String.ToUpper\(\)](#) and [String.ToLower\(\)](#) methods, because forcing a string to uppercase or lowercase is often used as a small normalization for comparing strings regardless of case. If so, consider using a case-insensitive comparison.

The [String.ToUpperInvariant](#) and [String.ToLowerInvariant](#) methods are also available. [ToUpperInvariant](#) is the standard way to normalize case. Comparisons made using [StringComparison.OrdinalIgnoreCase](#) are behaviorally the composition of two calls: calling [ToUpperInvariant](#) on both string arguments, and doing a comparison using [StringComparison.Ordinal](#).

Overloads are also available for converting to uppercase and lowercase in a specific culture, by passing a [CultureInfo](#) object that represents that culture to the method.

Char.ToUpper and Char.ToLower

Default interpretation: [StringComparison.CurrentCulture](#).

The [Char.ToUpper\(Char\)](#) and [Char.ToLower\(Char\)](#) methods work similarly to the [String.ToUpper\(\)](#) and [String.ToLower\(\)](#) methods described in the previous section.

String.StartsWith and String.EndsWith

Default interpretation: [StringComparison.CurrentCulture](#).

By default, both of these methods perform a culture-sensitive comparison. In particular, they may ignore non-printing characters.

String.IndexOf and String.LastIndexOf

Default interpretation: [StringComparison.CurrentCulture](#).

There's a lack of consistency in how the default overloads of these methods perform comparisons. All [String.IndexOf](#) and [String.LastIndexOf](#) methods that include a [Char](#) parameter perform an ordinal comparison, but the default [String.IndexOf](#) and [String.LastIndexOf](#) methods that include a [String](#) parameter perform a culture-sensitive comparison.

If you call the [String.IndexOf\(String\)](#) or [String.LastIndexOf\(String\)](#) method and pass it a string to locate in the current instance, we recommend that you call an overload that explicitly specifies the [StringComparison](#) type. The overloads that include a [Char](#) argument don't allow you to specify a [StringComparison](#) type.

Methods that perform string comparison indirectly

Some non-string methods that have string comparison as a central operation use the [StringComparer](#) type. The [StringComparer](#) class includes six static properties that return [StringComparer](#) instances whose [StringComparer.Compare](#) methods perform the following types of string comparisons:

- Culture-sensitive string comparisons using the current culture. This [StringComparer](#) object is returned by the [StringComparer.CurrentCulture](#) property.
- Case-insensitive comparisons using the current culture. This [StringComparer](#) object is returned by the [StringComparer.CurrentCultureIgnoreCase](#) property.
- Culture-insensitive comparisons using the word comparison rules of the invariant culture. This [StringComparer](#) object is returned by the

`StringComparer.InvariantCulture` property.

- Case-insensitive and culture-insensitive comparisons using the word comparison rules of the invariant culture. This `StringComparer` object is returned by the `StringComparer.InvariantCultureIgnoreCase` property.
- Ordinal comparison. This `StringComparer` object is returned by the `StringComparer.Ordinal` property.
- Case-insensitive ordinal comparison. This `StringComparer` object is returned by the `StringComparer.OrdinalIgnoreCase` property.

Array.Sort and Array.BinarySearch

Default interpretation: `StringComparison.CurrentCulture`.

When you store any data in a collection, or read persisted data from a file or database into a collection, switching the current culture can invalidate the invariants in the collection. The `Array.BinarySearch` method assumes that the elements in the array to be searched are already sorted. To sort any string element in the array, the `Array.Sort` method calls the `String.Compare` method to order individual elements. Using a culture-sensitive comparer can be dangerous if the culture changes between the time that the array is sorted and its contents are searched. For example, in the following code, storage and retrieval operate on the comparer that is provided implicitly by the `Thread.CurrentCulture` property. If the culture can change between the calls to `StoreNames` and `DoesNameExist`, and especially if the array contents are persisted somewhere between the two method calls, the binary search may fail.

C#

```
// Incorrect
string[] _storedNames;

public void StoreNames(string[] names)
{
    _storedNames = new string[names.Length];

    // Copy the array contents into a new array
    Array.Copy(names, _storedNames, names.Length);

    Array.Sort(_storedNames); // Line A
}

public bool DoesNameExist(string name) =>
    Array.BinarySearch(_storedNames, name) >= 0; // Line B
```

A recommended variation appears in the following example, which uses the same ordinal (culture-insensitive) comparison method both to sort and to search the array.

The change code is reflected in the lines labeled **Line A** and **Line B** in the two examples.

```
C#  
  
// Correct  
string[] _storedNames;  
  
public void StoreNames(string[] names)  
{  
    _storedNames = new string[names.Length];  
  
    // Copy the array contents into a new array  
    Array.Copy(names, _storedNames, names.Length);  
  
    Array.Sort(_storedNames, StringComparer.Ordinal); // Line A  
}  
  
public bool DoesNameExist(string name) =>  
    Array.BinarySearch(_storedNames, name, StringComparer.Ordinal) >= 0; //  
Line B
```

If this data is persisted and moved across cultures, and sorting is used to present this data to the user, you might consider using [StringComparison.InvariantCulture](#), which operates linguistically for better user output but is unaffected by changes in culture. The following example modifies the two previous examples to use the invariant culture for sorting and searching the array.

```
C#  
  
// Correct  
string[] _storedNames;  
  
public void StoreNames(string[] names)  
{  
    _storedNames = new string[names.Length];  
  
    // Copy the array contents into a new array  
    Array.Copy(names, _storedNames, names.Length);  
  
    Array.Sort(_storedNames, StringComparer.InvariantCulture); // Line A  
}  
  
public bool DoesNameExist(string name) =>  
    Array.BinarySearch(_storedNames, name, StringComparer.InvariantCulture)  
    >= 0; // Line B
```

Collections example: Hashtable constructor

Hashing strings provides a second example of an operation that is affected by the way in which strings are compared.

The following example instantiates a [Hashtable](#) object by passing it the [StringComparer](#) object that is returned by the [StringComparer.OrdinalIgnoreCase](#) property. Because a class [StringComparer](#) that is derived from [StringComparer](#) implements the [IEqualityComparer](#) interface, its [GetHashCode](#) method is used to compute the hash code of strings in the hash table.

C#

```
using System.IO;
using System.Collections;

const int InitialCapacity = 100;

Hashtable creationTimeByFile = new(InitialCapacity,
StringComparer.OrdinalIgnoreCase);
string directoryToProcess = Directory.GetCurrentDirectory();

// Fill the hash table
PopulateFileTable(directoryToProcess);

// Get some of the files and try to find them with upper cased names
foreach (var file in Directory.GetFiles(directoryToProcess))
    PrintCreationTime(file.ToUpper());


void PopulateFileTable(string directory)
{
    foreach (string file in Directory.GetFiles(directory))
        creationTimeByFile.Add(file, File.GetCreationTime(file));
}

void PrintCreationTime(string targetFile)
{
    object? dt = creationTimeByFile[targetFile];

    if (dt is DateTime value)
        Console.WriteLine($"File {targetFile} was created at time
{value}.");
    else
        Console.WriteLine($"File {targetFile} does not exist.");
}
```

See also

- [Globalization in .NET apps](#)

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

Best practices for displaying and persisting formatted data

Article • 01/27/2023

This article examines how formatted data, such as numeric data and date-and-time data, is handled for display and for storage.

When you develop with .NET, use culture-sensitive formatting to display non-string data, such as numbers and dates, in a user interface. Use formatting with the [invariant culture](#) to persist non-string data in string form. Do not use culture-sensitive formatting to persist numeric or date-and-time data in string form.

Display formatted data

When you display non-string data such as numbers and dates and times to users, format them by using the user's cultural settings. By default, the following all use the current culture in formatting operations:

- Interpolated strings supported by the [C#](#) and [Visual Basic](#) compilers.
- String concatenation operations that use the [C#](#) or [Visual Basic](#) concatenation operators or that call the [String.Concat](#) method directly.
- The [String.Format](#) method.
- The [ToString](#) methods of the numeric types and the date and time types.

To explicitly specify that a string should be formatted by using the conventions of a designated culture or the [invariant culture](#), you can do the following:

- When using the [String.Format](#) and [ToString](#) methods, call an overload that has a `provider` parameter, such as [String.Format\(IFormatProvider, String, Object\[\]\)](#) or [DateTime.ToString\(IFormatProvider\)](#), and pass it the [CultureInfo.CurrentCulture](#) property, a [CultureInfo](#) instance that represents the desired culture, or the [CultureInfo.InvariantCulture](#) property.
- For string concatenation, do not allow the compiler to perform any implicit conversions. Instead, perform an explicit conversion by calling a [ToString](#) overload that has a `provider` parameter. For example, the compiler implicitly uses the current culture when converting a [Double](#) value to a string in the following code:

C#

```
string concat1 = "The amount is " + 126.03 + ".";
Console.WriteLine(concat1);
```

Instead, you can explicitly specify the culture whose formatting conventions are used in the conversion by calling the [Double.ToString\(IFormatProvider\)](#) method, as the following code does:

C#

```
string concat2 = "The amount is " +
126.03.ToString(CultureInfo.InvariantCulture) + ".";
Console.WriteLine(concat2);
```

- For string interpolation, rather than assigning an interpolated string to a [String](#) instance, assign it to a [FormattableString](#). You can then call its [FormattableString.ToString\(\)](#) method to produce a result string that reflects the conventions of the current culture, or you can call the [FormattableString.ToString\(IFormatProvider\)](#) method to produce a result string that reflects the conventions of a specified culture.

You can also pass the formattable string to the static [FormattableString.Invariant](#) method to produce a result string that reflects the conventions of the invariant culture. The following example illustrates this approach. (The output from the example reflects a current culture of `en-US`.)

C#

```
using System;
using System.Globalization;

class Program
{
    static void Main()
    {
        Decimal value = 126.03m;
        FormattableString amount = $"The amount is {value:C}";
        Console.WriteLine(amount.ToString());
        Console.WriteLine(amount.ToString(new CultureInfo("fr-FR")));
        Console.WriteLine(FormattableString.Invariant(amount));
    }
}
// The example displays the following output:
//      The amount is $126.03
//      The amount is 126,03 €
//      The amount is #126.03
```

(!) Note

If you're using C# and formatting using the invariant culture, it's more performant to call `String.Create(IFormatProvider, DefaultInterpolatedStringHandler)` and pass `CultureInfo.InvariantCulture` for the first parameter. For more information, see [String interpolation in C# 10 and .NET 6](#).

Persist formatted data

You can persist non-string data either as binary data or as formatted data. If you choose to save it as formatted data, you should call a formatting method overload that includes a `provider` parameter and pass it the `CultureInfo.InvariantCulture` property. The invariant culture provides a consistent format for formatted data that is independent of culture and machine. In contrast, persisting data that is formatted by using cultures other than the invariant culture has a number of limitations:

- The data is likely to be unusable if it is retrieved on a system that has a different culture, or if the user of the current system changes the current culture and tries to retrieve the data.
- The properties of a culture on a specific computer can differ from standard values. At any time, a user can customize culture-sensitive display settings. Because of this, formatted data that is saved on a system may not be readable after the user customizes cultural settings. The portability of formatted data across computers is likely to be even more limited.
- International, regional, or national standards that govern the formatting of numbers or dates and times change over time, and these changes are incorporated into Windows operating system updates. When formatting conventions change, data that was formatted by using the previous conventions may become unreadable.

The following example illustrates the limited portability that results from using culture-sensitive formatting to persist data. The example saves an array of date and time values to a file. These are formatted by using the conventions of the English (United States) culture. After the application changes the current culture to French (Switzerland), it tries to read the saved values by using the formatting conventions of the current culture. The attempt to read two of the data items throws a `FormatException` exception, and the array of dates now contains two incorrect elements that are equal to `MinValue`.

```
using System;
using System.Globalization;
using System.IO;
using System.Text;
using System.Threading;

public class Example
{
    private static string filename = @".\dates.dat";

    public static void Main()
    {
        DateTime[] dates = { new DateTime(1758, 5, 6, 21, 26, 0),
                            new DateTime(1818, 5, 5, 7, 19, 0),
                            new DateTime(1870, 4, 22, 23, 54, 0),
                            new DateTime(1890, 9, 8, 6, 47, 0),
                            new DateTime(1905, 2, 18, 15, 12, 0) };
        // Write the data to a file using the current culture.
        WriteData(dates);
        // Change the current culture.
        Thread.CurrentThread.CurrentCulture =
CultureInfo.CreateSpecificCulture("fr-CH");
        // Read the data using the current culture.
        DateTime[] newDates = ReadData();
        foreach (var newDate in newDates)
            Console.WriteLine(newDate.ToString("g"));
    }

    private static void WriteData(DateTime[] dates)
    {
        StreamWriter sw = new StreamWriter(filename, false, Encoding.UTF8);
        for (int ctr = 0; ctr < dates.Length; ctr++) {
            sw.Write("{0}", dates[ctr].ToString("g",
CultureInfo.CurrentCulture));
            if (ctr < dates.Length - 1) sw.Write("|");
        }
        sw.Close();
    }

    private static DateTime[] ReadData()
    {
        bool exceptionOccurred = false;

        // Read file contents as a single string, then split it.
        StreamReader sr = new StreamReader(filename, Encoding.UTF8);
        string output = sr.ReadToEnd();
        sr.Close();

        string[] values = output.Split( new char[] { '|' } );
        DateTime[] newDates = new DateTime[values.Length];
        for (int ctr = 0; ctr < values.Length; ctr++) {
            try {
                newDates[ctr] = DateTime.Parse(values[ctr],
CultureInfo.CurrentCulture);
            }
        }
    }
}
```

```
        }
        catch (FormatException) {
            Console.WriteLine("Failed to parse {0}", values[ctr]);
            exceptionOccurred = true;
        }
    }
    if (exceptionOccurred) Console.WriteLine();
    return newDates;
}
}

// The example displays the following output:
//      Failed to parse 4/22/1870 11:54 PM
//      Failed to parse 2/18/1905 3:12 PM
//
//      05.06.1758 21:26
//      05.05.1818 07:19
//      01.01.0001 00:00
//      09.08.1890 06:47
//      01.01.0001 00:00
//      01.01.0001 00:00
```

However, if you replace the `CultureInfo.CurrentCulture` property with `CultureInfo.InvariantCulture` in the calls to `DateTime.ToString(String, IFormatProvider)` and `DateTime.Parse(String, IFormatProvider)`, the persisted date and time data is successfully restored, as the following output shows:

Console

```
06.05.1758 21:26
05.05.1818 07:19
22.04.1870 23:54
08.09.1890 06:47
18.02.1905 15:12
```

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

Behavior changes when comparing strings on .NET 5+

Article • 04/19/2023

.NET 5 introduces a runtime behavioral change where globalization APIs [use ICU by default](#) across all supported platforms. This is a departure from earlier versions of .NET Core and from .NET Framework, which utilize the operating system's national language support (NLS) functionality when running on Windows. For more information on these changes, including compatibility switches that can revert the behavior change, see [.NET globalization and ICU](#).

Reason for change

This change was introduced to unify .NET's globalization behavior across all supported operating systems. It also provides the ability for applications to bundle their own globalization libraries rather than depend on the OS's built-in libraries. For more information, see [the breaking change notification](#).

Behavioral differences

If you use functions like `string.IndexOf(string)` without calling the overload that takes a [StringComparison](#) argument, you might intend to perform an *ordinal* search, but instead you inadvertently take a dependency on culture-specific behavior. Since NLS and ICU implement different logic in their linguistic comparers, the results of methods like `string.IndexOf(string)` can return unexpected values.

This can manifest itself even in places where you aren't always expecting globalization facilities to be active. For example, the following code can produce a different answer depending on the current runtime.

C#

```
const string greeting = "He\0llo";
Console.WriteLine(${greeting.IndexOf("\0")});

// The snippet prints:
//
// '3' when running on .NET Core 2.x - 3.x (Windows)
// '0' when running on .NET 5 or later (Windows)
// '0' when running on .NET Core 2.x - 3.x or .NET 5 (non-Windows)
// '3' when running on .NET Core 2.x or .NET 5+ (in invariant mode)
```

```
string s = "Hello\r\nworld!";
int idx = s.IndexOf("\n");
Console.WriteLine(idx);

// The snippet prints:
//
// '6' when running on .NET Core 3.1
// '-1' when running on .NET 5 or .NET Core 3.1 (non-Windows OS)
// '-1' when running on .NET 5 (Windows 10 May 2019 Update or later)
// '6' when running on .NET 6+ (all Windows and non-Windows OS)
```

For more information, see [Globalization APIs use ICU libraries on Windows](#).

Guard against unexpected behavior

This section provides two options for dealing with unexpected behavior changes in .NET 5.

Enable code analyzers

[Code analyzers](#) can detect possibly buggy call sites. To help guard against any surprising behaviors, we recommend enabling .NET compiler platform (Roslyn) analyzers in your project. The analyzers help flag code that might inadvertently be using a linguistic comparer when an ordinal comparer was likely intended. The following rules should help flag these issues:

- [CA1307: Specify StringComparison for clarity](#)
- [CA1309: Use ordinal StringComparison](#)
- [CA1310: Specify StringComparison for correctness](#)

These specific rules aren't enabled by default. To enable them and show any violations as build errors, set the following properties in your project file:

XML

```
<PropertyGroup>
  <AnalysisMode>All</AnalysisMode>

  <WarningsAsErrors>$([WarningsAsErrors]);CA1307;CA1309;CA1310</WarningsAsErrors>
>
</PropertyGroup>
```

The following snippet shows examples of code that produces the relevant code analyzer warnings or errors.

C#

```
//  
// Potentially incorrect code - answer might vary based on locale.  
//  
string s = GetString();  
// Produces analyzer warning CA1310 for string; CA1307 matches on char ','  
int idx = s.IndexOf(",");  
Console.WriteLine(idx);  
  
//  
// Corrected code - matches the literal substring ",".  
//  
string s = GetString();  
int idx = s.IndexOf(",", StringComparison.Ordinal);  
Console.WriteLine(idx);  
  
//  
// Corrected code (alternative) - searches for the literal ',' character.  
//  
string s = GetString();  
int idx = s.IndexOf(',');  
Console.WriteLine(idx);
```

Similarly, when instantiating a sorted collection of strings or sorting an existing string-based collection, specify an explicit comparer.

C#

```
//  
// Potentially incorrect code - behavior might vary based on locale.  
//  
SortedSet<string> mySet = new SortedSet<string>();  
List<string> list = GetListOfStrings();  
list.Sort();  
  
//  
// Corrected code - uses ordinal sorting; doesn't vary by locale.  
//  
SortedSet<string> mySet = new SortedSet<string>(StringComparer.OrdinalIgnoreCase);  
List<string> list = GetListOfStrings();  
list.Sort(StringComparer.OrdinalIgnoreCase);
```

Revert back to NLS behaviors

To revert .NET 5+ applications back to older NLS behaviors when running on Windows, follow the steps in [.NET Globalization and ICU](#). This application-wide compatibility switch must be set at the application level. Individual libraries cannot opt-in or opt-out of this behavior.

💡 Tip

We strongly recommend you enable the [CA1307](#), [CA1309](#), and [CA1310](#) code analysis rules to help improve code hygiene and discover any existing latent bugs. For more information, see [Enable code analyzers](#).

Affected APIs

Most .NET applications won't encounter any unexpected behaviors due to the changes in .NET 5. However, due to the number of affected APIs and how foundational these APIs are to the wider .NET ecosystem, you should be aware of the potential for .NET 5 to introduce unwanted behaviors or to expose latent bugs that already exist in your application.

The affected APIs include:

- [System.String.Compare](#)
- [System.String.EndsWith](#)
- [System.String.IndexOf](#)
- [System.String.StartsWith](#)
- [System.String.ToLower](#)
- [System.String.ToLowerInvariant](#)
- [System.String.ToUpper](#)
- [System.String.ToUpperInvariant](#)
- [System.Globalization.TextInfo](#) (most members)
- [System.Globalization.CompareInfo](#) (most members)
- [System.Array.Sort](#) (when sorting arrays of strings)
- [System.Collections.Generic.List<T>.Sort\(\)](#) (when the list elements are strings)
- [System.Collections.Generic.SortedDictionary< TKey, TValue >](#) (when the keys are strings)
- [System.Collections.Generic.SortedList< TKey, TValue >](#) (when the keys are strings)
- [System.Collections.Generic.SortedSet< T >](#) (when the set contains strings)

⚠ Note

This is not an exhaustive list of affected APIs.

All of the above APIs use *linguistic* string searching and comparison using the thread's [current culture](#), by default. The differences between *linguistic* and *ordinal* search and comparison are called out in the [Ordinal vs. linguistic search and comparison](#).

Because ICU implements linguistic string comparisons differently from NLS, Windows-based applications that upgrade to .NET 5 from an earlier version of .NET Core or .NET Framework and that call one of the affected APIs may notice that the APIs begin exhibiting different behaviors.

Exceptions

- If an API accepts an explicit `StringComparison` or `CultureInfo` parameter, that parameter overrides the API's default behavior.
- `System.String` members where the first parameter is of type `char` (for example, `String.IndexOf(Char)`) use ordinal searching, unless the caller passes an explicit `StringComparison` argument that specifies `CurrentCulture[IgnoreCase]` or `InvariantCulture[IgnoreCase]`.

For a more detailed analysis of the default behavior of each `String` API, see the [Default search and comparison types](#) section.

Ordinal vs. linguistic search and comparison

Ordinal (also known as *non-linguistic*) search and comparison decomposes a string into its individual `char` elements and performs a char-by-char search or comparison. For example, the strings `"dog"` and `"dog"` compare as *equal* under an `Ordinal` comparer, since the two strings consist of the exact same sequence of chars. However, `"dog"` and `"Dog"` compare as *not equal* under an `Ordinal` comparer, because they don't consist of the exact same sequence of chars. That is, uppercase `'D'`'s code point `U+0044` occurs before lowercase `'d'`'s code point `U+0064`, resulting in `"Dog"` sorting before `"dog"`.

An `OrdinalIgnoreCase` comparer still operates on a char-by-char basis, but it eliminates case differences while performing the operation. Under an `OrdinalIgnoreCase` comparer, the char pairs `'d'` and `'D'` compare as *equal*, as do the char pairs `'á'` and `'Á'`. But the unaccented char `'a'` compares as *not equal* to the accented char `'á'`.

Some examples of this are provided in the following table:

| String 1 | String 2 | Ordinal comparison | OrdinalIgnoreCase comparison |
|-----------------------|-----------------------|--------------------|------------------------------|
| <code>"dog"</code> | <code>"dog"</code> | equal | equal |
| <code>"dog"</code> | <code>"Dog"</code> | not equal | equal |
| <code>"resume"</code> | <code>"résumé"</code> | not equal | not equal |

Unicode also allows strings to have several different in-memory representations. For example, an e-acute (é) can be represented in two possible ways:

- A single literal 'é' character (also written as '\u00E9').
- A literal unaccented 'e' character followed by a combining accent modifier character '\u0301'.

This means that the following *four* strings all display as "résumé", even though their constituent pieces are different. The strings use a combination of literal 'é' characters or literal unaccented 'e' characters plus the combining accent modifier '\u0301'.

- "r\u00E9sum\u00E9"
- "r\u00E9sume\u0301"
- "re\u0301sum\u00E9"
- "re\u0301sume\u0301"

Under an ordinal comparer, none of these strings compare as equal to each other. This is because they all contain different underlying char sequences, even though when they're rendered to the screen, they all look the same.

When performing a `string.IndexOf(..., StringComparison.OrdinalIgnoreCase)` operation, the runtime looks for an exact substring match. The results are as follows.

C#

```
Console.WriteLine("resume".IndexOf("e", StringComparison.OrdinalIgnoreCase)); // prints '1'
Console.WriteLine("r\u00E9sum\u00E9".IndexOf("e", StringComparison.OrdinalIgnoreCase)); // prints '-1'
Console.WriteLine("r\u00E9sume\u0301".IndexOf("e", StringComparison.OrdinalIgnoreCase)); // prints '5'
Console.WriteLine("re\u0301sum\u00E9".IndexOf("e", StringComparison.OrdinalIgnoreCase)); // prints '1'
Console.WriteLine("re\u0301sume\u0301".IndexOf("e", StringComparison.OrdinalIgnoreCase)); // prints '1'
Console.WriteLine("resume".IndexOf("E", StringComparison.OrdinalIgnoreCase)); // prints '1'
Console.WriteLine("r\u00E9sum\u00E9".IndexOf("E", StringComparison.OrdinalIgnoreCase)); // prints '-1'
Console.WriteLine("r\u00E9sume\u0301".IndexOf("E", StringComparison.OrdinalIgnoreCase)); // prints '5'
Console.WriteLine("re\u0301sum\u00E9".IndexOf("E", StringComparison.OrdinalIgnoreCase)); // prints '1'
Console.WriteLine("re\u0301sume\u0301".IndexOf("E", StringComparison.OrdinalIgnoreCase)); // prints '1'
```

Ordinal search and comparison routines are never affected by the current thread's culture setting.

Linguistic search and comparison routines decompose a string into *collation elements* and perform searches or comparisons on these elements. There's not necessarily a 1:1 mapping between a string's characters and its constituent collation elements. For example, a string of length 2 may consist of only a single collation element. When two strings are compared in a linguistic-aware fashion, the comparer checks whether the two strings' collation elements have the same semantic meaning, even if the string's literal characters are different.

Consider again the string "résumé" and its four different representations. The following table shows each representation broken down into its collation elements.

| String | As collation elements |
|--------------------|---|
| "r\u00E9sum\u00E9" | "r" + "\u00E9" + "s" + "u" + "m" + "\u00E9" |
| "r\u00E9sum\u0301" | "r" + "\u00E9" + "s" + "u" + "m" + "e\u0301" |
| "r\u0301sum\u00E9" | "r" + "e\u0301" + "s" + "u" + "m" + "\u00E9" |
| "r\u0301sum\u0301" | "r" + "e\u0301" + "s" + "u" + "m" + "e\u0301" |

A collation element corresponds loosely to what readers would think of as a single character or cluster of characters. It's conceptually similar to a [grapheme cluster](#) but encompasses a somewhat larger umbrella.

Under a linguistic comparer, exact matches aren't necessary. Collation elements are instead compared based on their semantic meaning. For example, a linguistic comparer treats the substrings "\u00E9" and "e\u0301" as equal since they both semantically mean "a lowercase e with an acute accent modifier." This allows the `IndexOf` method to match the substring "e\u0301" within a larger string that contains the semantically equivalent substring "\u00E9", as shown in the following code sample.

C#

```
Console.WriteLine("r\u00E9sum\u00E9.IndexOf("e")); // prints '-1' (not found)
Console.WriteLine("r\u00E9sum\u00E9.IndexOf("\u00E9")); // prints '1'
Console.WriteLine("\u00E9.IndexOf("e\u0301")); // prints '0'
```

As a consequence of this, two strings of different lengths may compare as equal if a linguistic comparison is used. Callers should take care not to special-case logic that deals with string length in such scenarios.

Culture-aware search and comparison routines are a special form of linguistic search and comparison routines. Under a culture-aware comparer, the concept of a collation element is extended to include information specific to the specified culture.

For example, [in the Hungarian alphabet](#), when the two characters <dz> appear back-to-back, they are considered their own unique letter distinct from either <d> or <z>. This means that when <dz> is seen in a string, a Hungarian culture-aware comparer treats it as a single collation element.

| String | As collation elements | Remarks |
|--------|-----------------------|--|
| "endz" | "e" + "n" + "d" + "z" | (using a standard linguistic comparer) |
| "endz" | "e" + "n" + "dz" | (using a Hungarian culture-aware comparer) |

When using a Hungarian culture-aware comparer, this means that the string "endz" *does not* end with the substring "z", as <dz> and <z> are considered collation elements with different semantic meaning.

C#

```
// Set thread culture to Hungarian
CultureInfo.CurrentCulture = CultureInfo.GetCultureInfo("hu-HU");
Console.WriteLine("endz".EndsWith("z")); // Prints 'False'

// Set thread culture to invariant culture
CultureInfo.CurrentCulture = CultureInfo.InvariantCulture;
Console.WriteLine("endz".EndsWith("z")); // Prints 'True'
```

ⓘ Note

- Behavior: Linguistic and culture-aware comparers can undergo behavioral adjustments from time to time. Both ICU and the older Windows NLS facility are updated to account for how world languages change. For more information, see the blog post [Locale \(culture\) data churn](#). The *Ordinal* comparer's behavior will never change since it performs exact bitwise searching and comparison. However, the *OrdinalIgnoreCase* comparer's behavior may change as Unicode grows to encompass more character sets and corrects omissions in existing casing data.
- Usage: The comparers `StringComparison.InvariantCulture` and `StringComparison.InvariantCultureIgnoreCase` are linguistic comparers that are not culture-aware. That is, these comparers understand concepts such as

the accented character é having multiple possible underlying representations, and that all such representations should be treated equal. But non-culture-aware linguistic comparers won't contain special handling for <dz> as distinct from <d> or <z>, as shown above. They also won't special-case characters like the German Eszett (ß).

.NET also offers the *invariant globalization mode*. This opt-in mode disables code paths that deal with linguistic search and comparison routines. In this mode, all operations use *Ordinal* or *OrdinalIgnoreCase* behaviors, regardless of what `CultureInfo` or `StringComparison` argument the caller provides. For more information, see [Runtime configuration options for globalization](#) and [.NET Core Globalization Invariant Mode](#).

For more information, see [Best practices for comparing strings in .NET](#).

Security implications

If your app uses an affected API for filtering, we recommend enabling the CA1307 and CA1309 code analysis rules to help locate places where a linguistic search may have inadvertently been used instead of an ordinal search. Code patterns like the following may be susceptible to security exploits.

C#

```
//  
// THIS SAMPLE CODE IS INCORRECT.  
// DO NOT USE IT IN PRODUCTION.  
  
public bool ContainsHtmlSensitiveCharacters(string input)  
{  
    if (input.IndexOf("<") >= 0) { return true; }  
    if (input.IndexOf("&") >= 0) { return true; }  
    return false;  
}
```

Because the `string.IndexOf(string)` method uses a linguistic search by default, it's possible for a string to contain a literal '`<`' or '`&`' character and for the `string.IndexOf(string)` routine to return `-1`, indicating that the search substring was not found. Code analysis rules CA1307 and CA1309 flag such call sites and alert the developer that there's a potential problem.

Default search and comparison types

The following table lists the default search and comparison types for various string and string-like APIs. If the caller provides an explicit `CultureInfo` or `StringComparison` parameter, that parameter will be honored over any default.

| API | Default behavior | Remarks |
|--------------------------------------|------------------|--|
| <code>string.Compare</code> | CurrentCulture | |
| <code>string.CompareTo</code> | CurrentCulture | |
| <code>string.Contains</code> | Ordinal | |
| <code>string.EndsWith</code> | Ordinal | (when the first parameter is a <code>char</code>) |
| <code>string.EndsWith</code> | CurrentCulture | (when the first parameter is a <code>string</code>) |
| <code>string.Equals</code> | Ordinal | |
| <code>string.GetHashCode</code> | Ordinal | |
| <code>string.IndexOf</code> | Ordinal | (when the first parameter is a <code>char</code>) |
| <code>string.IndexOf</code> | CurrentCulture | (when the first parameter is a <code>string</code>) |
| <code>string.IndexOfAny</code> | Ordinal | |
| <code>string.LastIndexOf</code> | Ordinal | (when the first parameter is a <code>char</code>) |
| <code>string.LastIndexOf</code> | CurrentCulture | (when the first parameter is a <code>string</code>) |
| <code>string.LastIndexOfAny</code> | Ordinal | |
| <code>string.Replace</code> | Ordinal | |
| <code>string.Split</code> | Ordinal | |
| <code>string.StartsWith</code> | Ordinal | (when the first parameter is a <code>char</code>) |
| <code>string.StartsWith</code> | CurrentCulture | (when the first parameter is a <code>string</code>) |
| <code>string.ToLower</code> | CurrentCulture | |
| <code>string.ToLowerInvariant</code> | InvariantCulture | |
| <code>string.ToUpper</code> | CurrentCulture | |
| <code>string.ToUpperInvariant</code> | InvariantCulture | |
| <code>string.Trim</code> | Ordinal | |
| <code>string.TrimEnd</code> | Ordinal | |

| API | Default behavior | Remarks |
|-------------------------------|------------------|---------|
| <code>string.TrimStart</code> | Ordinal | |
| <code>string == string</code> | Ordinal | |
| <code>string != string</code> | Ordinal | |

Unlike `string` APIs, all `MemoryExtensions` APIs perform *Ordinal* searches and comparisons by default, with the following exceptions.

| API | Default behavior | Remarks |
|--|-------------------------------|--|
| <code>MemoryExtensions.ToLower</code> | <code>CurrentCulture</code> | (when passed a null <code>CultureInfo</code> argument) |
| <code>MemoryExtensions.ToLowerInvariant</code> | <code>InvariantCulture</code> | |
| <code>MemoryExtensions.ToUpper</code> | <code>CurrentCulture</code> | (when passed a null <code>CultureInfo</code> argument) |
| <code>MemoryExtensions.ToUpperInvariant</code> | <code>InvariantCulture</code> | |

A consequence is that when converting code from consuming `string` to consuming `ReadOnlySpan<char>`, behavioral changes may be introduced inadvertently. An example of this follows.

```
C#
string str = GetString();
if (str.StartsWith("Hello")) { /* do something */ } // this is a CULTURE-AWARE (linguistic) comparison

ReadOnlySpan<char> span = s.AsSpan();
if (span.StartsWith("Hello")) { /* do something */ } // this is an ORDINAL (non-linguistic) comparison
```

The recommended way to address this is to pass an explicit `StringComparison` parameter to these APIs. The code analysis rules CA1307 and CA1309 can assist with this.

```
C#
string str = GetString();
if (str.StartsWith("Hello", StringComparison.OrdinalIgnoreCase)) { /* do something */ }
} // ordinal comparison
```

```
ReadOnlySpan<char> span = s.AsSpan();
if (span.StartsWith("Hello", StringComparison.Ordinal)) { /* do something */
} // ordinal comparison
```

See also

- [Globalization breaking changes](#)
- [Best practices for comparing strings in .NET](#)
- [How to compare strings in C#](#)
- [.NET globalization and ICU](#)
- [Ordinal vs. culture-sensitive string operations](#)
- [Overview of .NET source code analysis](#)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

Basic string operations in .NET

Article • 09/15/2021

Applications often respond to users by constructing messages based on user input. For example, it is not uncommon for websites to respond to a newly logged-on user with a specialized greeting that includes the user's name.

Several methods in the [System.String](#) and [System.Text.StringBuilder](#) classes allow you to dynamically construct custom strings to display in your user interface. These methods also help you perform a number of basic string operations like creating new strings from arrays of bytes, comparing the values of strings, and modifying existing strings.

Related sections

[Type Conversion in .NET](#)

Describes how to convert one type into another type.

[Formatting Types](#)

Describes how to format strings using format specifiers.

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

Creating New Strings in .NET

Article • 09/15/2021

.NET allows strings to be created using simple assignment, and also overloads a class constructor to support string creation using a number of different parameters. .NET also provides several methods in the [System.String](#) class that create new string objects by combining several strings, arrays of strings, or objects.

Creating Strings Using Assignment

The easiest way to create a new [String](#) object is simply to assign a string literal to a [String](#) object.

Creating Strings Using a Class Constructor

You can use overloads of the [String](#) class constructor to create strings from character arrays. You can also create a new string by duplicating a particular character a specified number of times.

Methods that Return Strings

The following table lists several useful methods that return new string objects.

| Method | Use |
|-------------------------------|--|
| name | |
| String.Format | Builds a formatted string from a set of input objects. |
| String.Concat | Builds strings from two or more strings. |
| String.Join | Builds a new string by combining an array of strings. |
| String.Insert | Builds a new string by inserting a string into the specified index of an existing string. |
| String.CopyTo | Copies specified characters in a string into a specified position in an array of characters. |

Format

You can use the **String.Format** method to create formatted strings and concatenate strings representing multiple objects. This method automatically converts any passed object into a string. For example, if your application must display an **Int32** value and a **DateTime** value to the user, you can easily construct a string to represent these values using the **Format** method. For information about formatting conventions used with this method, see the section on [composite formatting](#).

The following example uses the **Format** method to create a string that uses an integer variable.

C#

```
int numberOfFleas = 12;
string miscInfo = String.Format("Your dog has {0} fleas. " +
                                 "It is time to get a flea collar. " +
                                 "The current universal date is: {1:u}.",
                                 numberOffleas, DateTime.Now);
Console.WriteLine(miscInfo);
// The example displays the following output:
//      Your dog has 12 fleas. It is time to get a flea collar.
//      The current universal date is: 2008-03-28 13:31:40Z.
```

In this example, **DateTime.Now** displays the current date and time in a manner specified by the culture associated with the current thread.

Concat

The **String.Concat** method can be used to easily create a new string object from two or more existing objects. It provides a language-independent way to concatenate strings. This method accepts any class that derives from **System.Object**. The following example creates a string from two existing string objects and a separating character.

C#

```
string helloString1 = "Hello";
string helloString2 = "World!";
Console.WriteLine(String.Concat(helloString1, ' ', helloString2));
// The example displays the following output:
//      Hello World!
```

Join

The **String.Join** method creates a new string from an array of strings and a separator string. This method is useful if you want to concatenate multiple strings together,

making a list perhaps separated by a comma.

The following example uses a space to bind a string array.

C#

```
string[] words = {"Hello", "and", "welcome", "to", "my" , "world!"};
Console.WriteLine(String.Join(" ", words));
// The example displays the following output:
//      Hello and welcome to my world!
```

Insert

The **String.Insert** method creates a new string by inserting a string into a specified position in another string. This method uses a zero-based index. The following example inserts a string into the fifth index position of `MyString` and creates a new string with this value.

C#

```
string sentence = "Once a time.";
Console.WriteLine(sentence.Insert(4, " upon"));
// The example displays the following output:
//      Once upon a time.
```

CopyTo

The **String.CopyTo** method copies portions of a string into an array of characters. You can specify both the beginning index of the string and the number of characters to be copied. This method takes the source index, an array of characters, the destination index, and the number of characters to copy. All indexes are zero-based.

The following example uses the **CopyTo** method to copy the characters of the word "Hello" from a string object to the first index position of an array of characters.

C#

```
string greeting = "Hello World!";
char[] charArray = {'W','h','e','r','e'};
Console.WriteLine("The original character array: {0}", new
string(charArray));
greeting.CopyTo(0, charArray,0 ,5);
Console.WriteLine("The new character array: {0}", new string(charArray));
// The example displays the following output:
```

```
//      The original character array: Where  
//      The new character array: Hello
```

See also

- [Basic String Operations](#)
- [Composite Formatting](#)

 [Collaborate with us on GitHub](#)

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback [here](#).

 [Open a documentation issue](#)

 [Provide product feedback](#)

Trim and remove characters from strings in .NET

Article • 10/04/2022

If you're parsing a sentence into individual words, you might end up with words that have blank spaces (also called white spaces) on either end of the word. In this situation, you can use one of the trim methods in the `System.String` class to remove any number of spaces or other characters from a specified position in the string. The following table describes the available trim methods:

| Method name | Use |
|-------------------------------|--|
| <code>String.Trim</code> | Removes white spaces or characters specified in an array of characters from the beginning and end of a string. |
| <code>String.TrimEnd</code> | Removes characters specified in an array of characters from the end of a string. |
| <code>String.TrimStart</code> | Removes characters specified in an array of characters from the beginning of a string. |
| <code>String.Remove</code> | Removes a specified number of characters from a specified index position in a string. |

Trim

You can easily remove white spaces from both ends of a string by using the `String.Trim` method, as shown in the following example:

C#

```
string MyString = " Big   ";
Console.WriteLine("Hello{0}World!", MyString);
string TrimString = MyString.Trim();
Console.WriteLine("Hello{0}World!", TrimString);
//      The example displays the following output:
//          Hello Big   World!
//          HelloBigWorld!
```

You can also remove characters that you specify in a character array from the beginning and end of a string. The following example removes white-space characters, periods, and asterisks:

C#

```
using System;

public class Example
{
    public static void Main()
    {
        String header = "* A Short String. *";
        Console.WriteLine(header);
        Console.WriteLine(header.Trim( new Char[] { ' ', '*', '.' } ));
    }
}

// The example displays the following output:
//      * A Short String. *
//      A Short String
```

TrimEnd

The `String.TrimEnd` method removes characters from the end of a string, creating a new string object. An array of characters is passed to this method to specify the characters to be removed. The order of the elements in the character array doesn't affect the trim operation. The trim stops when a character not specified in the array is found.

The following example removes the last letters of a string using the `TrimEnd` method. In this example, the position of the `'r'` character and the `'w'` character are reversed to illustrate that the order of characters in the array doesn't matter. Notice that this code removes the last word of `MyString` plus part of the first.

C#

```
string MyString = "Hello World!";
char[] MyChar = {'r','o','W','l','d','!', ' '};
string NewString = MyString.TrimEnd(MyChar);
Console.WriteLine(NewString);
```

This code displays `He` to the console.

The following example removes the last word of a string using the `TrimEnd` method. In this code, a comma follows the word `Hello` and because the comma isn't specified in the array of characters to trim, the trim ends at the comma.

C#

```
string MyString = "Hello, World!";
char[] MyChar = {'r','o','W','l','d','!', ' '};
```

```
string NewString = MyString.TrimEnd(MyChar);
Console.WriteLine(NewString);
```

This code displays `Hello,` to the console.

TrimStart

The `String.TrimStart` method is similar to the `String.TrimEnd` method except that it creates a new string by removing characters from the beginning of an existing string object. An array of characters is passed to the `TrimStart` method to specify the characters to be removed. As with the `TrimEnd` method, the order of the elements in the character array doesn't affect the trim operation. The trim stops when a character not specified in the array is found.

The following example removes the first word of a string. In this example, the position of the `'l'` character and the `'H'` character are reversed to illustrate that the order of characters in the array doesn't matter.

C#

```
string MyString = "Hello World!";
char[] MyChar = {'e', 'H', 'l', 'o', ' ' };
string NewString = MyString.TrimStart(MyChar);
Console.WriteLine(NewString);
```

This code displays `World!` to the console.

Remove

The `String.Remove` method removes a specified number of characters that begin at a specified position in an existing string. This method assumes a zero-based index.

The following example removes 10 characters from a string beginning at position five of a zero-based index of the string.

C#

```
string MyString = "Hello Beautiful World!";
Console.WriteLine(MyString.Remove(5,10));
// The example displays the following output:
//      Hello World!
```

Replace

You can also remove a specified character or substring from a string by calling the [String.Replace\(String, String\)](#) method and specifying an empty string ([String.Empty](#)) as the replacement. The following example removes all commas from a string:

C#

```
using System;

public class Example
{
    public static void Main()
    {
        String phrase = "a cold, dark night";
        Console.WriteLine("Before: {0}", phrase);
        phrase = phrase.Replace(",", "");
        Console.WriteLine("After: {0}", phrase);
    }
}

// The example displays the following output:
//      Before: a cold, dark night
//      After: a cold dark night
```

See also

- [Basic String Operations](#)

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

 .NET feedback

The .NET documentation is open source. Provide feedback [here](#).

 Open a documentation issue

 Provide product feedback

Padding Strings in .NET

Article • 09/15/2021

Use one of the following [String](#) methods to create a new string that consists of an original string that is padded with leading or trailing characters to a specified total length. The padding character can be a space or a specified character. The resulting string appears to be either right-aligned or left-aligned. If the original string's length is already equal to or greater than the desired total length, the padding methods return the original string unchanged; for more information, see the **Returns** sections of the two overloads of the [String.PadLeft](#) and [String.PadRight](#) methods.

| Method name | Use |
|---------------------------------|---|
| String.PadLeft | Pads a string with leading characters to a specified total length. |
| String.PadRight | Pads a string with trailing characters to a specified total length. |

PadLeft

The [String.PadLeft](#) method creates a new string by concatenating enough leading pad characters to an original string to achieve a specified total length. The [String.PadLeft\(Int32\)](#) method uses white space as the padding character and the [String.PadLeft\(Int32, Char\)](#) method enables you to specify your own padding character.

The following code example uses the [PadLeft](#) method to create a new string that is twenty characters long. The example displays "-----Hello World!" to the console.

C#

```
string MyString = "Hello World!";
Console.WriteLine(MyString.PadLeft(20, ' - '));
```

PadRight

The [String.PadRight](#) method creates a new string by concatenating enough trailing pad characters to an original string to achieve a specified total length. The [String.PadRight\(Int32\)](#) method uses white space as the padding character and the [String.PadRight\(Int32, Char\)](#) method enables you to specify your own padding character.

The following code example uses the [PadRight](#) method to create a new string that is twenty characters long. The example displays "Hello World!-----" to the console.

C#

```
string MyString = "Hello World!";
Console.WriteLine(MyString.PadRight(20, '-'));
```

See also

- [Basic String Operations](#)

 [Collaborate with us on GitHub](#)

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback [here](#).

 [Open a documentation issue](#)

 [Provide product feedback](#)

Compare strings in .NET

Article • 09/15/2021

.NET provides several methods to compare the values of strings. The following table lists and describes the value-comparison methods.

| Method name | Use |
|---------------------------------------|---|
| String.Compare | Compares the values of two strings. Returns an integer value. |
| String.CompareOrdinal | Compares two strings without regard to local culture. Returns an integer value. |
| String.CompareTo | Compares the current string object to another string. Returns an integer value. |
| String.StartsWith | Determines whether a string begins with the string passed. Returns a Boolean value. |
| String.EndsWith | Determines whether a string ends with the string passed. Returns a Boolean value. |
| String.Contains | Determines whether a character or string occurs within another string. Returns a Boolean value. |
| String.Equals | Determines whether two strings are the same. Returns a Boolean value. |
| String.IndexOf | Returns the index position of a character or string, starting from the beginning of the string you are examining. Returns an integer value. |
| String.LastIndexOf | Returns the index position of a character or string, starting from the end of the string you are examining. Returns an integer value. |

Compare method

The static [String.Compare](#) method provides a thorough way of comparing two strings. This method is culturally aware. You can use this function to compare two strings or substrings of two strings. Additionally, overloads are provided that regard or disregard case and cultural variance. The following table shows the three integer values that this method might return.

| Return value | Condition |
|--------------------|--|
| A negative integer | The first string precedes the second string in the sort order. -or- |
| A positive integer | The first string follows the second string in the sort order. |

| Return value | Condition |
|--------------------|---|
| | The first string is <code>null</code> . |
| 0 | The first string and the second string are equal. -or- Both strings are <code>null</code> . |
| A positive integer | The first string follows the second string in the sort order. |
| -or- | -or- |
| 1 | The second string is <code>null</code> . |

ⓘ Important

The [String.Compare](#) method is primarily intended for use when ordering or sorting strings. You should not use the [String.Compare](#) method to test for equality (that is, to explicitly look for a return value of 0 with no regard for whether one string is less than or greater than the other). Instead, to determine whether two strings are equal, use the [String.Equals\(String, String, StringComparison\)](#) method.

The following example uses the [String.Compare](#) method to determine the relative values of two strings.

C#

```
string string1 = "Hello World!";
Console.WriteLine(String.Compare(string1, "Hello World?"));
```

This example displays `-1` to the console.

The preceding example is culture-sensitive by default. To perform a culture-insensitive string comparison, use an overload of the [String.Compare](#) method that allows you to specify the culture to use by supplying a *culture* parameter. For an example that demonstrates how to use the [String.Compare](#) method to perform a culture-insensitive comparison, see [Culture-insensitive string comparisons](#).

CompareOrdinal method

The `String.CompareOrdinal` method compares two string objects without considering the local culture. The return values of this method are identical to the values returned by the `Compare` method in the previous table.

ⓘ Important

The `String.CompareOrdinal` method is primarily intended for use when ordering or sorting strings. You should not use the `String.CompareOrdinal` method to test for equality (that is, to explicitly look for a return value of 0 with no regard for whether one string is less than or greater than the other). Instead, to determine whether two strings are equal, use the `String.Equals(String, String, StringComparison)` method.

The following example uses the `CompareOrdinal` method to compare the values of two strings.

C#

```
string string1 = "Hello World!";
Console.WriteLine(String.CompareOrdinal(string1, "hello world!"));
```

This example displays `-32` to the console.

CompareTo method

The `String.CompareTo` method compares the string that the current string object encapsulates to another string or object. The return values of this method are identical to the values returned by the `String.Compare` method in the previous table.

ⓘ Important

The `String.CompareTo` method is primarily intended for use when ordering or sorting strings. You should not use the `String.CompareTo` method to test for equality (that is, to explicitly look for a return value of 0 with no regard for whether one string is less than or greater than the other). Instead, to determine whether two strings are equal, use the `String.Equals(String, String, StringComparison)` method.

The following example uses the `String.CompareTo` method to compare the `string1` object to the `string2` object.

C#

```
string string1 = "Hello World";
string string2 = "Hello World!";
int MyInt = string1.CompareTo(string2);
Console.WriteLine( MyInt );
```

This example displays `-1` to the console.

All overloads of the [String.CompareTo](#) method perform culture-sensitive and case-sensitive comparisons by default. No overloads of this method are provided that allow you to perform a culture-insensitive comparison. For code clarity, we recommend that you use the [String.Compare](#) method instead, specifying [CultureInfo.CurrentCulture](#) for culture-sensitive operations or [CultureInfo.InvariantCulture](#) for culture-insensitive operations. For examples that demonstrate how to use the [String.Compare](#) method to perform both culture-sensitive and culture-insensitive comparisons, see [Performing Culture-Insensitive String Comparisons](#).

Equals method

The [String.Equals](#) method can easily determine if two strings are the same. This case-sensitive method returns a `true` or `false` Boolean value. It can be used from an existing class, as illustrated in the next example. The following example uses the [Equals](#) method to determine whether a string object contains the phrase "Hello World".

C#

```
string string1 = "Hello World";
Console.WriteLine(string1.Equals("Hello World"));
```

This example displays `True` to the console.

This method can also be used as a static method. The following example compares two string objects using a static method.

C#

```
string string1 = "Hello World";
string string2 = "Hello World";
Console.WriteLine(String.Equals(string1, string2));
```

This example displays `True` to the console.

StartsWith and EndsWith methods

You can use the [String.StartsWith](#) method to determine whether a string object begins with the same characters that encompass another string. This case-sensitive method returns `true` if the current string object begins with the passed string and `false` if it does not. The following example uses this method to determine if a string object begins with "Hello".

C#

```
string string1 = "Hello World";
Console.WriteLine(string1.StartsWith("Hello"));
```

This example displays `True` to the console.

The [String.EndsWith](#) method compares a passed string to the characters that exist at the end of the current string object. It also returns a Boolean value. The following example checks the end of a string using the `EndsWith` method.

C#

```
string string1 = "Hello World";
Console.WriteLine(string1.EndsWith("Hello"));
```

This example displays `False` to the console.

IndexOf and LastIndexOf methods

You can use the [String.IndexOf](#) method to determine the position of the first occurrence of a particular character within a string. This case-sensitive method starts counting from the beginning of a string and returns the position of a passed character using a zero-based index. If the character cannot be found, a value of `-1` is returned.

The following example uses the `IndexOf` method to search for the first occurrence of the '`l`' character in a string.

C#

```
string string1 = "Hello World";
Console.WriteLine(string1.IndexOf('l'));
```

This example displays `2` to the console.

The `String.LastIndexOf` method is similar to the `String.IndexOf` method except that it returns the position of the last occurrence of a particular character within a string. It is case-sensitive and uses a zero-based index.

The following example uses the `LastIndexOf` method to search for the last occurrence of the '`'l'`' character in a string.

C#

```
string string1 = "Hello World";
Console.WriteLine(string1.LastIndexOf('l'));
```

This example displays `9` to the console.

Both methods are useful when used in conjunction with the `String.Remove` method. You can use either the `IndexOf` or `LastIndexOf` methods to retrieve the position of a character, and then supply that position to the `Remove` method in order to remove a character or a word that begins with that character.

See also

- [Best practices for using strings in .NET](#)
- [Basic string operations](#)
- [Perform culture-insensitive string operations](#)
- [Sorting weight tables](#) - used by .NET Framework and .NET Core 1.0-3.1 on Windows
- [Default Unicode collation element table](#) - used by .NET 5 on all platforms, and by .NET Core on Linux and macOS

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback [here](#).

 [Open a documentation issue](#)

 [Provide product feedback](#)

Change case in .NET

Article • 06/08/2022

If you write an application that accepts input from a user, you can never be sure what case (upper or lower) they will use to enter the data. Often, you want strings to be cased consistently, particularly if you are displaying them in the user interface. The following table describes three case-changing methods. The first two methods provide an overload that accepts a culture.

| Method name | Use |
|--------------------------------------|---|
| String.ToUpper | Converts all characters in a string to uppercase. |
| String.ToLower | Converts all characters in a string to lowercase. |
| TextInfo.ToTitleCase | Converts a string to title case. |

⚠ Warning

The [String.ToUpper](#) and [String.ToLower](#) methods should not be used to convert strings in order to compare them or test them for equality. For more information, see the [Compare strings of mixed case](#) section.

Compare strings of mixed case

To compare strings of mixed case to determine their ordering, call one of the overloads of the [String.CompareTo](#) method with a `comparisonType` parameter, and provide a value of either [StringComparison.CurrentCultureIgnoreCase](#), [StringComparison.InvariantCultureIgnoreCase](#), or [StringComparison.OrdinalIgnoreCase](#) for the `comparisonType` argument. For a comparison using a specific culture other than the current culture, call an overload of the [String.CompareTo](#) method with both a `culture` and `options` parameter, and provide a value of [CompareOptions.IgnoreCase](#) as the `options` argument.

To compare strings of mixed case to determine whether they're equal, call one of the overloads of the [String.Equals](#) method with a `comparisonType` parameter, and provide a value of either [StringComparison.CurrentCultureIgnoreCase](#), [StringComparison.InvariantCultureIgnoreCase](#), or [StringComparison.OrdinalIgnoreCase](#) for the `comparisonType` argument.

For more information, see [Best practices for using strings](#).

ToUpper method

The [String.ToUpper](#) method changes all characters in a string to uppercase. The following example converts the string "Hello World!" from mixed case to uppercase.

C#

```
string properString = "Hello World!";
Console.WriteLine(properString.ToUpper());
// This example displays the following output:
//      HELLO WORLD!
```

The preceding example is culture-sensitive by default; it applies the casing conventions of the current culture. To perform a culture-insensitive case change or to apply the casing conventions of a particular culture, use the [String.ToUpper\(CultureInfo\)](#) method overload and supply a value of [CultureInfo.InvariantCulture](#) or a [System.Globalization.CultureInfo](#) object that represents the specified culture to the `culture` parameter. For an example that demonstrates how to use the [ToUpper](#) method to perform a culture-insensitive case change, see [Perform culture-insensitive case changes](#).

ToLower method

The [String.ToLower](#) method is similar to the previous method, but instead converts all the characters in a string to lowercase. The following example converts the string "Hello World!" to lowercase.

C#

```
string properString = "Hello World!";
Console.WriteLine(properString.ToLower());
// This example displays the following output:
//      hello world!
```

The preceding example is culture-sensitive by default; it applies the casing conventions of the current culture. To perform a culture-insensitive case change or to apply the casing conventions of a particular culture, use the [String.ToLower\(CultureInfo\)](#) method overload and supply a value of [CultureInfo.InvariantCulture](#) or a [System.Globalization.CultureInfo](#) object that represents the specified culture to the `culture` parameter. For an example that demonstrates how to use the

[ToLower\(CultureInfo\)](#) method to perform a culture-insensitive case change, see [Perform culture-insensitive case changes](#).

ToTitleCase method

The [TextInfo.ToTitleCase](#) converts the first character of each word to uppercase and the remaining characters to lowercase. However, words that are entirely uppercase are assumed to be acronyms and are not converted.

The [TextInfo.ToTitleCase](#) method is culture-sensitive; that is, it uses the casing conventions of a particular culture. In order to call the method, you first retrieve the [TextInfo](#) object that represents the casing conventions of the particular culture from the [CultureInfo.TextInfo](#) property of a particular culture.

The following example passes each string in an array to the [TextInfo.ToTitleCase](#) method. The strings include proper title strings as well as acronyms. The strings are converted to title case by using the casing conventions of the English (United States) culture.

C#

```
using System;
using System.Globalization;

public class Example
{
    public static void Main()
    {
        string[] values = { "a tale of two cities", "gROWL to the rescue",
                           "inside the US government", "sports and MLB
baseball",
                           "The Return of Sherlock Holmes", "UNICEF and
children"};

        TextInfo ti = CultureInfo.CurrentCulture.TextInfo;
        foreach (var value in values)
            Console.WriteLine("{0} --> {1}", value, ti.ToTitleCase(value));
    }
}

// The example displays the following output:
//   a tale of two cities --> A Tale Of Two Cities
//   gROWL to the rescue --> Growl To The Rescue
//   inside the US government --> Inside The US Government
//   sports and MLB baseball --> Sports And MLB Baseball
//   The Return of Sherlock Holmes --> The Return Of Sherlock Holmes
//   UNICEF and children --> UNICEF And Children
```

Note that although it is culture-sensitive, the `TextInfo.ToTitleCase` method does not provide linguistically correct casing rules. For instance, in the previous example, the method converts "a tale of two cities" to "A Tale Of Two Cities". However, the linguistically correct title casing for the en-US culture is "A Tale of Two Cities."

See also

- [Basic String Operations](#)
- [Perform culture-insensitive string operations](#)

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

Extract substrings from a string

Article • 09/15/2021

This article covers some different techniques for extracting parts of a string.

- Use the [Split method](#) when the substrings you want are separated by a known delimiting character (or characters).
- [Regular expressions](#) are useful when the string conforms to a fixed pattern.
- Use the [IndexOf](#) and [Substring methods](#) in conjunction when you don't want to extract *all* of the substrings in a string.

String.Split method

[String.Split](#) provides a handful of overloads to help you break up a string into a group of substrings based on one or more delimiting characters that you specify. You can choose to limit the total number of substrings in the final result, trim white-space characters from substrings, or exclude empty substrings.

The following examples show three different overloads of `String.Split()`. The first example calls the [Split\(Char\[\]\)](#) overload without passing any separator characters. When you don't specify any delimiting characters, `String.Split()` uses default delimiters, which are white-space characters, to split up the string.

C#

```
string s = "You win some. You lose some.";  
  
string[] subs = s.Split();  
  
foreach (string sub in subs)  
{  
    Console.WriteLine($"Substring: {sub}");  
}  
  
// This example produces the following output:  
//  
// Substring: You  
// Substring: win  
// Substring: some.  
// Substring: You  
// Substring: lose  
// Substring: some.
```

As you can see, the period characters (.) are included in two of the substrings. If you want to exclude the period characters, you can add the period character as an additional delimiting character. The next example shows how to do this.

C#

```
string s = "You win some. You lose some.";  
  
string[] subs = s.Split(' ', '.');  
  
foreach (string sub in subs)  
{  
    Console.WriteLine($"Substring: {sub}");  
}  
  
// This example produces the following output:  
//  
// Substring: You  
// Substring: win  
// Substring: some  
// Substring:  
// Substring: You  
// Substring: lose  
// Substring: some  
// Substring:
```

The periods are gone from the substrings, but now two extra empty substrings have been included. These empty substring represent the substring between the word and the period that follows it. To omit empty substrings from the resulting array, you can call the [Split\(Char\[\], StringSplitOptions\)](#) overload and specify [StringSplitOptions.RemoveEmptyEntries](#) for the `options` parameter.

C#

```
string s = "You win some. You lose some.";  
char[] separators = new char[] { ' ', '.' };  
  
string[] subs = s.Split(separators, StringSplitOptions.RemoveEmptyEntries);  
  
foreach (string sub in subs)  
{  
    Console.WriteLine($"Substring: {sub}");  
}  
  
// This example produces the following output:  
//  
// Substring: You  
// Substring: win  
// Substring: some  
// Substring: You
```

```
// Substring: lose  
// Substring: some
```

Regular expressions

If your string conforms to a fixed pattern, you can use a regular expression to extract and handle its elements. For example, if strings take the form "*number operand number*", you can use a [regular expression](#) to extract and handle the string's elements. Here's an example:

C#

```
String[] expressions = { "16 + 21", "31 * 3", "28 / 3",  
                        "42 - 18", "12 * 7",  
                        "2, 4, 6, 8" };  
String pattern = @"(\d+)\s+([-*/])\s+(\d+)";  
  
foreach (string expression in expressions)  
{  
    foreach (System.Text.RegularExpressions.Match m in  
        System.Text.RegularExpressions.Regex.Matches(expression, pattern))  
    {  
        int value1 = Int32.Parse(m.Groups[1].Value);  
        int value2 = Int32.Parse(m.Groups[3].Value);  
        switch (m.Groups[2].Value)  
        {  
            case "+":  
                Console.WriteLine("{0} = {1}", m.Value, value1 + value2);  
                break;  
            case "-":  
                Console.WriteLine("{0} = {1}", m.Value, value1 - value2);  
                break;  
            case "*":  
                Console.WriteLine("{0} = {1}", m.Value, value1 * value2);  
                break;  
            case "/":  
                Console.WriteLine("{0} = {1:N2}", m.Value, value1 / value2);  
                break;  
        }  
    }  
}  
  
// The example displays the following output:  
//      16 + 21 = 37  
//      31 * 3 = 93  
//      28 / 3 = 9.33  
//      42 - 18 = 24  
//      12 * 7 = 84
```

The regular expression pattern `(\d+)\s+([-+*/])\s+(\d+)` is defined like this:

| Pattern | Description |
|-----------------------|--|
| <code>(\d+)</code> | Match one or more decimal digits. This is the first capturing group. |
| <code>\s+</code> | Match one or more white-space characters. |
| <code>([-+*/])</code> | Match an arithmetic operator sign (+, -, *, or /). This is the second capturing group. |
| <code>\s+</code> | Match one or more white-space characters. |
| <code>(\d+)</code> | Match one or more decimal digits. This is the third capturing group. |

You can also use a regular expression to extract substrings from a string based on a pattern rather than a fixed set of characters. This is a common scenario when either of these conditions occurs:

- One or more of the delimiter characters does not *always* serve as a delimiter in the [String](#) instance.
- The sequence and number of delimiter characters is variable or unknown.

For example, the [Split](#) method cannot be used to split the following string, because the number of `\n` (newline) characters is variable, and they don't always serve as delimiters.

```
text

[This is captured\n{text.}]\n\n[\n[This is more captured text.]\n]
\n[Some more captured text:\n    Option1\n    Option2][Terse text.]
```

A regular expression can split this string easily, as the following example shows.

```
C#


String input = "[This is captured\n{text.}]\n\n[\n[This is more captured text.]\n]
\n[Some more captured text:\n    Option1" +
    "\n    Option2][Terse text.]";
String pattern = @"\[(^\[\]]+)\]";
int ctr = 0;

foreach (System.Text.RegularExpressions.Match m in
    System.Text.RegularExpressions.Regex.Matches(input, pattern))
{
    Console.WriteLine("{0}: {1}", ++ctr, m.Groups[1].Value);
}

// The example displays the following output:
```

```
//      1: This is captured
//      text.
//      2: This is more captured text.
//      3: Some more captured text:
//          Option1
//          Option2
//      4: Terse text.
```

The regular expression pattern `\[(\[^[\]]+\)]` is defined like this:

| Pattern | Description |
|---------------------------|--|
| <code>\[</code> | Match an opening bracket. |
| <code>([^\\[\\]]+)</code> | Match any character that is not an opening or a closing bracket one or more times. This is the first capturing group. |
| <code>\]</code> | Match a closing bracket. |

The [Regex.Split](#) method is almost identical to [String.Split](#), except that it splits a string based on a regular expression pattern instead of a fixed character set. For example, the following example uses the [Regex.Split](#) method to split a string that contains substrings delimited by various combinations of hyphens and other characters.

C#

```
String input = "abacus -- alabaster - * - atrium -+- " +
              "any -* - actual - + - armoire - - alarm";
String pattern = @"\s-\s?[*]? \s?- \s";
String[] elements = System.Text.RegularExpressions.Regex.Split(input,
pattern);

foreach (string element in elements)
    Console.WriteLine(element);

// The example displays the following output:
//      abacus
//      alabaster
//      atrium
//      any
//      actual
//      armoire
//      alarm
```

The regular expression pattern `\s-\s?[*]? \s?- \s` is defined like this:

| Pattern | Description |
|------------------|---|
| <code>\s-</code> | Match a white-space character followed by a hyphen. |

| Pattern | Description |
|---------|--|
| \s? | Match zero or one white-space character. |
| [+*]? | Match zero or one occurrence of either the + or * character. |
| \s? | Match zero or one white-space character. |
| -\s | Match a hyphen followed by a white-space character. |

String.IndexOf and String.Substring methods

If you aren't interested in all of the substrings in a string, you might prefer to work with one of the string comparison methods that returns the index at which the match begins. You can then call the [Substring](#) method to extract the substring that you want. The string comparison methods include:

- [IndexOf](#), which returns the zero-based index of the first occurrence of a character or string in a string instance.
- [IndexOfAny](#), which returns the zero-based index in the current string instance of the first occurrence of any character in a character array.
- [LastIndexOf](#), which returns the zero-based index of the last occurrence of a character or string in a string instance.
- [LastIndexOfAny](#), which returns a zero-based index in the current string instance of the last occurrence of any character in a character array.

The following example uses the [IndexOf](#) method to find the periods in a string. It then uses the [Substring](#) method to return full sentences.

C#

```
String s = "This is the first sentence in a string. " +
           "More sentences will follow. For example, " +
           "this is the third sentence. This is the " +
           "fourth. And this is the fifth and final " +
           "sentence.";
var sentences = new List<String>();
int start = 0;
int position;

// Extract sentences from the string.
do
{
    position = s.IndexOf('.', start);
```

```
if (position >= 0)
{
    sentences.Add(s.Substring(start, position - start + 1).Trim());
    start = position + 1;
}
} while (position > 0);

// Display the sentences.
foreach (var sentence in sentences)
    Console.WriteLine(sentence);

// The example displays the following output:
//      This is the first sentence in a string.
//      More sentences will follow.
//      For example, this is the third sentence.
//      This is the fourth.
//      And this is the fifth and final sentence.
```

See also

- Basic string operations in .NET
- .NET regular expressions
- How to parse strings using String.Split in C#

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

Using the StringBuilder Class in .NET

Article • 09/15/2021

The [String](#) object is immutable. Every time you use one of the methods in the [System.String](#) class, you create a new string object in memory, which requires a new allocation of space for that new object. In situations where you need to perform repeated modifications to a string, the overhead associated with creating a new [String](#) object can be costly. The [System.Text.StringBuilder](#) class can be used when you want to modify a string without creating a new object. For example, using the [StringBuilder](#) class can boost performance when concatenating many strings together in a loop.

Importing the System.Text Namespace

The [StringBuilder](#) class is found in the [System.Text](#) namespace. To avoid having to provide a fully qualified type name in your code, you can import the [System.Text](#) namespace:

```
C#
```

```
using System;
using System.Text;
```

Instantiating a StringBuilder Object

You can create a new instance of the [StringBuilder](#) class by initializing your variable with one of the overloaded constructor methods, as illustrated in the following example.

```
C#
```

```
StringBuilder myStringBuilder = new StringBuilder("Hello World!");
```

Setting the Capacity and Length

Although the [StringBuilder](#) is a dynamic object that allows you to expand the number of characters in the string that it encapsulates, you can specify a value for the maximum number of characters that it can hold. This value is called the capacity of the object and should not be confused with the length of the string that the current [StringBuilder](#) holds. For example, you might create a new instance of the [StringBuilder](#) class with the string "Hello", which has a length of 5, and you might specify that the object has a

maximum capacity of 25. When you modify the [StringBuilder](#), it does not reallocate size for itself until the capacity is reached. When this occurs, the new space is allocated automatically and the capacity is doubled. You can specify the capacity of the [StringBuilder](#) class using one of the overloaded constructors. The following example specifies that the `myStringBuilder` object can be expanded to a maximum of 25 spaces.

C#

```
StringBuilder myStringBuilder = new StringBuilder("Hello World!", 25);
```

Additionally, you can use the read/write [Capacity](#) property to set the maximum length of your object. The following example uses the [Capacity](#) property to define the maximum object length.

C#

```
myStringBuilder.Capacity = 25;
```

The [EnsureCapacity](#) method can be used to check the capacity of the current [StringBuilder](#). If the capacity is greater than the passed value, no change is made; however, if the capacity is smaller than the passed value, the current capacity is changed to match the passed value.

The [Length](#) property can also be viewed or set. If you set the [Length](#) property to a value that is greater than the [Capacity](#) property, the [Capacity](#) property is automatically changed to the same value as the [Length](#) property. Setting the [Length](#) property to a value that is less than the length of the string within the current [StringBuilder](#) shortens the string.

Modifying the [StringBuilder](#) String

The following table lists the methods you can use to modify the contents of a [StringBuilder](#).

| Method name | Use |
|--|--|
| StringBuilder.Append | Appends information to the end of the current StringBuilder . |
| StringBuilder.AppendFormat | Replaces a format specifier passed in a string with formatted text. |
| StringBuilder.Insert | Inserts a string or object into the specified index of the current StringBuilder . |

| Method name | Use |
|---------------------------------------|---|
| StringBuilder.Remove | Removes a specified number of characters from the current StringBuilder . |
| StringBuilder.Replace | Replaces all occurrences of a specified character or string in the current StringBuilder with another specified character or string. |

Append

The **Append** method can be used to add text or a string representation of an object to the end of a string represented by the current **StringBuilder**. The following example initializes a **StringBuilder** to "Hello World" and then appends some text to the end of the object. Space is allocated automatically as needed.

C#

```
StringBuilder myStringBuilder = new StringBuilder("Hello World!");
myStringBuilder.Append(" What a beautiful day.");
Console.WriteLine(myStringBuilder);
// The example displays the following output:
//      Hello World! What a beautiful day.
```

AppendFormat

The [StringBuilder.AppendFormat](#) method adds text to the end of the **StringBuilder** object. It supports the composite formatting feature (for more information, see [Composite Formatting](#)) by calling the [IFormattable](#) implementation of the object or objects to be formatted. Therefore, it accepts the standard format strings for numeric, date and time, and enumeration values, the custom format strings for numeric and date and time values, and the format strings defined for custom types. (For information about formatting, see [Formatting Types](#).) You can use this method to customize the format of variables and append those values to a **StringBuilder**. The following example uses the [AppendFormat](#) method to place an integer value formatted as a currency value at the end of a **StringBuilder** object.

C#

```
int MyInt = 25;
StringBuilder myStringBuilder = new StringBuilder("Your total is ");
myStringBuilder.AppendFormat("{0:C} ", MyInt);
Console.WriteLine(myStringBuilder);
// The example displays the following output:
//      Your total is $25.00
```

Insert

The [Insert](#) method adds a string or object to a specified position in the current [StringBuilder](#) object. The following example uses this method to insert a word into the sixth position of a [StringBuilder](#) object.

C#

```
StringBuilder myStringBuilder = new StringBuilder("Hello World!");
myStringBuilder.Insert(6,"Beautiful ");
Console.WriteLine(myStringBuilder);
// The example displays the following output:
//      Hello Beautiful World!
```

Remove

You can use the [Remove](#) method to remove a specified number of characters from the current [StringBuilder](#) object, beginning at a specified zero-based index. The following example uses the [Remove](#) method to shorten a [StringBuilder](#) object.

C#

```
StringBuilder myStringBuilder = new StringBuilder("Hello World!");
myStringBuilder.Remove(5,7);
Console.WriteLine(myStringBuilder);
// The example displays the following output:
//      Hello
```

Replace

The [Replace](#) method can be used to replace characters within the [StringBuilder](#) object with another specified character. The following example uses the [Replace](#) method to search a [StringBuilder](#) object for all instances of the exclamation point character (!) and replace them with the question mark character (?).

C#

```
StringBuilder myStringBuilder = new StringBuilder("Hello World!");
myStringBuilder.Replace('!', '?');
Console.WriteLine(myStringBuilder);
// The example displays the following output:
//      Hello World?
```

Converting a `StringBuilder` Object to a `String`

You must convert the `StringBuilder` object to a `String` object before you can pass the string represented by the `StringBuilder` object to a method that has a `String` parameter or display it in the user interface. You do this conversion by calling the `StringBuilder.ToString` method. The following example calls a number of `StringBuilder` methods and then calls the `StringBuilder.ToString()` method to display the string.

C#

```
using System;
using System.Text;

public class Example
{
    public static void Main()
    {
        StringBuilder sb = new StringBuilder();
        bool flag = true;
        string[] spellings = { "recieve", "receeve", "receive" };
        sb.AppendFormat("Which of the following spellings is {0}:", flag);
        sb.AppendLine();
        for (int ctr = 0; ctr <= spellings.GetUpperBound(0); ctr++) {
            sb.AppendFormat(" {0}. {1}", ctr, spellings[ctr]);
            sb.AppendLine();
        }
        sb.AppendLine();
        Console.WriteLine(sb.ToString());
    }
}
// The example displays the following output:
//      Which of the following spellings is True:
//          0. recieve
//          1. receeve
//          2. receive
```

See also

- [System.Text.StringBuilder](#)
- [Basic String Operations](#)
- [Formatting Types](#)



Collaborate with us on
GitHub

.NET

.NET feedback

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

How to: Perform Basic String Manipulations in .NET

Article • 09/15/2021

The following example uses some of the methods discussed in the [Basic String Operations](#) topics to construct a class that performs string manipulations in a manner that might be found in a real-world application. The `MailToData` class stores the name and address of an individual in separate properties and provides a way to combine the `City`, `State`, and `Zip` fields into a single string for display to the user. Furthermore, the class allows the user to enter the city, state, and zip code information as a single string. The application automatically parses the single string and enters the proper information into the corresponding property.

For simplicity, this example uses a console application with a command-line interface.

Example

C#

```
using System;

class MainClass
{
    static void Main()
    {
        MailToData MyData = new MailToData();

        Console.Write("Enter Your Name: ");
        MyData.Name = Console.ReadLine();
        Console.Write("Enter Your Address: ");
        MyData.Address = Console.ReadLine();
        Console.Write("Enter Your City, State, and ZIP Code separated by
spaces: ");
        MyData.CityStateZip = Console.ReadLine();
        Console.WriteLine();

        if (MyData.Validated) {
            Console.WriteLine("Name: {0}", MyData.Name);
            Console.WriteLine("Address: {0}", MyData.Address);
            Console.WriteLine("City: {0}", MyData.City);
            Console.WriteLine("State: {0}", MyData.State);
            Console.WriteLine("Zip: {0}", MyData.Zip);

            Console.WriteLine("\nThe following address will be used:");
            Console.WriteLine(MyData.Address);
            Console.WriteLine(MyData.CityStateZip);
```

```
        }
    }

public class MailToData
{
    string name = "";
    string address = "";
    string citystatezip = "";
    string city = "";
    string state = "";
    string zip = "";
    bool parseSucceeded = false;

    public string Name
    {
        get{return name;}
        set{name = value;}
    }

    public string Address
    {
        get{return address;}
        set{address = value;}
    }

    public string CityStateZip
    {
        get {
            return String.Format("{0}, {1} {2}", city, state, zip);
        }
        set {
            citystatezip = value.Trim();
            ParseCityStateZip();
        }
    }

    public string City
    {
        get{return city;}
        set{city = value;}
    }

    public string State
    {
        get{return state;}
        set{state = value;}
    }

    public string Zip
    {
        get{return zip;}
        set{zip = value;}
    }
}
```

```

public bool Validated
{
    get { return parseSucceeded; }
}

private void ParseCityStateZip()
{
    string msg = "";
    const string msgEnd = "\nYou must enter spaces between city, state,
and zip code.\n";

    // Throw a FormatException if the user did not enter the necessary
    // spaces
    // between elements.
    try
    {
        // City may consist of multiple words, so we'll have to parse the
        // string from right to left starting with the zip code.
        int zipIndex = citystatezip.LastIndexOf(" ");
        if (zipIndex == -1) {
            msg = "\nCannot identify a zip code." + msgEnd;
            throw new FormatException(msg);
        }
        zip = citystatezip.Substring(zipIndex + 1);

        int stateIndex = citystatezip.LastIndexOf(" ", zipIndex - 1);
        if (stateIndex == -1) {
            msg = "\nCannot identify a state." + msgEnd;
            throw new FormatException(msg);
        }
        state = citystatezip.Substring(stateIndex + 1, zipIndex -
stateIndex - 1);
        state = state.ToUpper();

        city = citystatezip.Substring(0, stateIndex);
        if (city.Length == 0) {
            msg = "\nCannot identify a city." + msgEnd;
            throw new FormatException(msg);
        }
        parseSucceeded = true;
    }
    catch (FormatException ex)
    {
        Console.WriteLine(ex.Message);
    }
}

private string ReturnCityStateZip()
{
    // Make state uppercase.
    state = state.ToUpper();

    // Put the value of city, state, and zip together in the proper
    // manner.
    string MyCityStateZip = String.Concat(city, " ", state, " ", zip);
}

```

```
        return MyCityStateZip;  
    }  
}
```

When the preceding code is executed, the user is asked to enter their name and address. The application places the information in the appropriate properties and displays the information back to the user, creating a single string that displays the city, state, and zip code information.

See also

- [Basic String Operations](#)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

Open a documentation issue

Provide product feedback

Parse strings in .NET

Article • 09/15/2021

A *parsing* operation converts a string that represents a .NET base type into that base type. For example, a parsing operation is used to convert a string to a floating-point number or to a date-and-time value. The method most commonly used to perform a parsing operation is the `Parse` method. Because parsing is the reverse operation of formatting (which involves converting a base type into its string representation), many of the same rules and conventions apply. Just as formatting uses an object that implements the `IFormatProvider` interface to provide culture-sensitive formatting information, parsing also uses an object that implements the `IFormatProvider` interface to determine how to interpret a string representation. For more information, see [Format types](#).

In This Section

[Parsing Numeric Strings](#)

Describes how to convert strings into .NET numeric types.

[Parsing Date and Time Strings](#)

Describes how to convert strings into .NET `DateTime` types.

[Parsing Other Strings](#)

Describes how to convert strings into `Char`, `Boolean`, and `Enum` types.

Related Sections

[Formatting Types](#)

Describes basic formatting concepts like format specifiers and format providers.

[Type Conversion in .NET](#)

Describes how to convert types.

 Collaborate with us on
GitHub

The source for this content can
be found on GitHub, where you
can also create and review



.NET feedback

The .NET documentation is open
source. Provide feedback here.

 [Open a documentation issue](#)

issues and pull requests. For more information, see [our contributor guide](#).

 [Provide product feedback](#)

Parsing numeric strings in .NET

Article • 10/04/2022

All numeric types have two static parsing methods, `Parse` and `TryParse`, that you can use to convert the string representation of a number into a numeric type. These methods enable you to parse strings that were produced by using the format strings documented in [Standard Numeric Format Strings](#) and [Custom Numeric Format Strings](#). By default, the `Parse` and `TryParse` methods can successfully convert strings that contain integral decimal digits only to integer values. They can successfully convert strings that contain integral and fractional decimal digits, group separators, and a decimal separator to floating-point values. The `Parse` method throws an exception if the operation fails, whereas the `TryParse` method returns `false`.

ⓘ Note

Starting in .NET 7, the numeric types in .NET also implement the `System.IParseable<TSelf>` interface, which defines the `IParsable<TSelf>.Parse` and `IParsable<TSelf>.TryParse` methods.

Parsing and format providers

Typically, the string representations of numeric values differ by culture. Elements of numeric strings, such as currency symbols, group (or thousands) separators, and decimal separators, all vary by culture. Parsing methods either implicitly or explicitly use a format provider that recognizes these culture-specific variations. If no format provider is specified in a call to the `Parse` or `TryParse` method, the format provider associated with the current culture (the `NumberFormatInfo` object returned by the `NumberFormatInfo.CurrentInfo` property) is used.

A format provider is represented by an `IFormatProvider` implementation. This interface has a single member, the `GetFormat` method, whose single parameter is a `Type` object that represents the type to be formatted. This method returns the object that provides formatting information. .NET supports the following two `IFormatProvider` implementations for parsing numeric strings:

- A `CultureInfo` object whose `CultureInfo.GetFormat` method returns a `NumberFormatInfo` object that provides culture-specific formatting information.

- A [NumberFormatInfo](#) object whose [NumberFormatInfo.GetFormat](#) method returns itself.

The following example tries to convert each string in an array to a [Double](#) value. It first tries to parse the string by using a format provider that reflects the conventions of the English (United States) culture. If this operation throws a [FormatException](#), it tries to parse the string by using a format provider that reflects the conventions of the French (France) culture.

C#

```
using System;
using System.Globalization;

public class Example
{
    public static void Main()
    {
        string[] values = { "1,304.16", "$1,456.78", "1,094", "152",
                            "123,45 €", "1 304,16", "Ae9f" };
        double number;
        CultureInfo culture = null;

        foreach (string value in values) {
            try {
                culture = CultureInfo.CreateSpecificCulture("en-US");
                number = Double.Parse(value, culture);
                Console.WriteLine("{0}: {1} --> {2}", culture.Name, value,
number);
            }
            catch (FormatException) {
                Console.WriteLine("{0}: Unable to parse '{1}'.",
culture.Name, value);
                culture = CultureInfo.CreateSpecificCulture("fr-FR");
                try {
                    number = Double.Parse(value, culture);
                    Console.WriteLine("{0}: {1} --> {2}", culture.Name, value,
number);
                }
                catch (FormatException) {
                    Console.WriteLine("{0}: Unable to parse '{1}'.",
culture.Name, value);
                }
            }
            Console.WriteLine();
        }
    }
}

// The example displays the following output:
//      en-US: 1,304.16 --> 1304.16
//
//      en-US: Unable to parse '$1,456.78'.
```

```
//    fr-FR: Unable to parse '$1,456.78'.
//
//    en-US: 1,094 --> 1094
//
//    en-US: 152 --> 152
//
//    en-US: Unable to parse '123,45 €'.
//    fr-FR: Unable to parse '123,45 €'.
//
//    en-US: Unable to parse '1 304,16'.
//    fr-FR: 1 304,16 --> 1304.16
//
//    en-US: Unable to parse 'Ae9f'.
//    fr-FR: Unable to parse 'Ae9f'.
```

Parsing and NumberStyles Values

The style elements (such as white space, group separators, and decimal separator) that the parse operation can handle are defined by a [NumberStyles](#) enumeration value. By default, strings that represent integer values are parsed by using the [NumberStyles.Integer](#) value, which permits only numeric digits, leading and trailing white space, and a leading sign. Strings that represent floating-point values are parsed using a combination of the [NumberStyles.Float](#) and [NumberStyles.AllowThousands](#) values; this composite style permits decimal digits along with leading and trailing white space, a leading sign, a decimal separator, a group separator, and an exponent. By calling an overload of the [Parse](#) or [TryParse](#) method that includes a parameter of type [NumberStyles](#) and setting one or more [NumberStyles](#) flags, you can control the style elements that can be present in the string for the parse operation to succeed.

For example, a string that contains a group separator can't be converted to an [Int32](#) value by using the [Int32.Parse\(String\)](#) method. However, the conversion succeeds if you use the [NumberStyles.AllowThousands](#) flag, as the following example illustrates.

C#

```
using System;
using System.Globalization;

public class Example
{
    public static void Main()
    {
        string value = "1,304";
        int number;
        IFormatProvider provider = CultureInfo.CreateSpecificCulture("en-US");
        if (Int32.TryParse(value, out number))
            Console.WriteLine("{0} --> {1}", value, number);
```

```

        else
            Console.WriteLine("Unable to convert '{0}'", value);

        if (Int32.TryParse(value, NumberStyles.Integer |
NumberStyles.AllowThousands,
                    provider, out number))
            Console.WriteLine("{0} --> {1}", value, number);
        else
            Console.WriteLine("Unable to convert '{0}'", value);
    }
}

// The example displays the following output:
//      Unable to convert '1,304'
//      1,304 --> 1304

```

Warning

The parse operation always uses the formatting conventions of a particular culture. If you do not specify a culture by passing a [CultureInfo](#) or [NumberFormatInfo](#) object, the culture associated with the current thread is used.

The following table lists the members of the [NumberStyles](#) enumeration and describes the effect that they have on the parsing operation.

| NumberStyles value | Effect on the string to be parsed |
|---|--|
| NumberStyles.None | Only numeric digits are permitted. |
| NumberStyles.AllowDecimalPoint | The decimal separator and fractional digits are permitted. For integer values, only zero is permitted as a fractional digit. Valid decimal separators are determined by the NumberFormatInfo.NumberDecimalSeparator or NumberFormatInfo.CurrencyDecimalSeparator property. |
| NumberStyles.AllowExponent | The "e" or "E" character can be used to indicate exponential notation. For more information, see NumberStyles . |
| NumberStyles.AllowLeadingWhite | Leading white space is permitted. |
| NumberStyles.AllowTrailingWhite | Trailing white space is permitted. |
| NumberStyles.AllowLeadingSign | A positive or negative sign can precede numeric digits. |
| NumberStyles.AllowTrailingSign | A positive or negative sign can follow numeric digits. |
| NumberStyles.AllowParentheses | Parentheses can be used to indicate negative values. |

| NumberStyles value | Effect on the string to be parsed |
|----------------------------------|---|
| NumberStyles.AllowThousands | The group separator is permitted. The group separator character is determined by the NumberFormatInfo.NumberGroupSeparator or NumberFormatInfo.CurrencyGroupSeparator property. |
| NumberStyles.AllowCurrencySymbol | The currency symbol is permitted. The currency symbol is defined by the NumberFormatInfo.CurrencySymbol property. |
| NumberStyles.AllowHexSpecifier | The string to be parsed is interpreted as a hexadecimal number. It can include the hexadecimal digits 0-9, A-F, and a-f. This flag can be used only to parse integer values. |

In addition, the [NumberStyles](#) enumeration provides the following composite styles, which include multiple [NumberStyles](#) flags.

| Composite NumberStyles value | Includes members |
|------------------------------|---|
| NumberStyles.Integer | Includes the NumberStyles.AllowLeadingWhite , NumberStyles.AllowTrailingWhite , and NumberStyles.AllowLeadingSign styles. This is the default style used to parse integer values. |
| NumberStyles.Number | Includes the NumberStyles.AllowLeadingWhite , NumberStyles.AllowTrailingWhite , NumberStyles.AllowLeadingSign , NumberStyles.AllowTrailingSign , NumberStyles.AllowDecimalPoint , and NumberStyles.AllowThousands styles. |
| NumberStyles.Float | Includes the NumberStyles.AllowLeadingWhite , NumberStyles.AllowTrailingWhite , NumberStyles.AllowLeadingSign , NumberStyles.AllowDecimalPoint , and NumberStyles.AllowExponent styles. |
| NumberStyles.Currency | Includes all styles except NumberStyles.AllowExponent and NumberStyles.AllowHexSpecifier . |
| NumberStyles.Any | Includes all styles except NumberStyles.AllowHexSpecifier . |
| NumberStyles.HexNumber | Includes the NumberStyles.AllowLeadingWhite , NumberStyles.AllowTrailingWhite , and NumberStyles.AllowHexSpecifier styles. |

Parsing and Unicode Digits

The Unicode standard defines code points for digits in various writing systems. For example, code points from U+0030 to U+0039 represent the basic Latin digits 0 through 9, code points from U+09E6 to U+09EF represent the Bangla digits 0 through 9, and code points from U+FF10 to U+FF19 represent the Fullwidth digits 0 through 9. However, the only numeric digits recognized by parsing methods are the basic Latin digits 0-9 with code points from U+0030 to U+0039. If a numeric parsing method is passed a string that contains any other digits, the method throws a [FormatException](#).

The following example uses the [Int32.Parse](#) method to parse strings that consist of digits in different writing systems. As the output from the example shows, the attempt to parse the basic Latin digits succeeds, but the attempt to parse the Fullwidth, Arabic-Indic, and Bangla digits fails.

C#

```
using System;

public class Example
{
    public static void Main()
    {
        string value;
        // Define a string of basic Latin digits 1-5.
        value = "\u0031\u0032\u0033\u0034\u0035";
        ParseDigits(value);

        // Define a string of Fullwidth digits 1-5.
        value = "\uFF11\uFF12\uFF13\uFF14\uFF15";
        ParseDigits(value);

        // Define a string of Arabic-Indic digits 1-5.
        value = "\u0661\u0662\u0663\u0664\u0665";
        ParseDigits(value);

        // Define a string of Bangla digits 1-5.
        value = "\u09e7\u09e8\u09e9\u09ea\u09eb";
        ParseDigits(value);
    }

    static void ParseDigits(string value)
    {
        try {
            int number = Int32.Parse(value);
            Console.WriteLine("{0} --> {1}", value, number);
        }
        catch (FormatException) {
            Console.WriteLine("Unable to parse '{0}'.", value);
        }
    }
}

// The example displays the following output:
```

```
//      '12345' --> 12345
//      Unable to parse '1 2 3 4 5'.
//      Unable to parse '۱۲۳۴۵'.
//      Unable to parse '۱۲۳۸۶'.
```

See also

- [NumberStyles](#)
- [Parsing Strings](#)
- [Formatting Types](#)

 [Collaborate with us on GitHub](#)

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

Parse date and time strings in .NET

Article • 10/04/2022

Parsing strings to convert them to [DateTime](#) objects requires you to specify information about how the dates and times are represented as text. Different cultures use different orders for day, month, and year. Some time representations use a 24-hour clock, others specify "AM" and "PM." Some applications need only the date. Others need only the time. Still others need to specify both the date and time. The methods that convert strings to [DateTime](#) objects enable you to provide detailed information about the formats you expect and the elements of a date and time your application needs. There are three subtasks to correctly converting text into a [DateTime](#):

1. You must specify the expected format of the text representing a date and time.
2. You can specify the culture for the format of a date time.
3. You can specify how missing components in the text representation are set in the date and time.

The [Parse](#) and [TryParse](#) methods convert many common representations of a date and time. The [ParseExact](#) and [TryParseExact](#) methods convert a string representation that conforms to the pattern specified by a date and time format string. For more information, see the articles on [standard date and time format strings](#) and [custom date and time format strings](#).

The current [DateTimeFormatInfo](#) object provides more control over how text should be interpreted as a date and time. Properties of a [DateTimeFormatInfo](#) describe the date and time separators, the names of months, days, and eras, and the format for the "AM" and "PM" designations. The [CultureInfo](#) returned by [CultureInfo.CurrentCulture](#) has a [CultureInfo.DateTimeFormat](#) property that represents the current culture. If you want a specific culture or custom settings, you specify the [IFormatProvider](#) parameter of a parsing method. For the [IFormatProvider](#) parameter, specify a [CultureInfo](#) object, which represents a culture, or a [DateTimeFormatInfo](#) object.

The text representing a date or time might be missing some information. For example, most people would assume the date "March 12" represents the current year. Similarly, "March 2018" represents the month of March in the year 2018. Text representing time often does only include hours, minutes, and an AM/PM designation. Parsing methods handle this missing information by using reasonable defaults:

- When only the time is present, the date portion uses the current date.
- When only the date is present, the time portion is midnight.
- When the year isn't specified in a date, the current year is used.

- When the day of the month isn't specified, the first day of the month is used.

If the date is present in the string, it must include the month and one of the day or year. If the time is present, it must include the hour, and either the minutes or the AM/PM designator.

You can specify the [NoCurrentDateDefault](#) constant to override these defaults. When you use that constant, any missing year, month, or day properties are set to the value `1`. The [last example](#) using [Parse](#) demonstrates this behavior.

In addition to a date and a time component, the string representation of a date and time can include an offset that indicates how much the time differs from Coordinated Universal Time (UTC). For example, the string "2/14/2007 5:32:00 -7:00" defines a time that is seven hours earlier than UTC. If an offset is omitted from the string representation of a time, parsing returns a [DateTime](#) object with its [Kind](#) property set to [DateTimeKind.Unspecified](#). If an offset is specified, parsing returns a [DateTime](#) object with its [Kind](#) property set to [DateTimeKind.Local](#). Its value is also adjusted to the local time zone of your machine. You can modify this behavior by using a [DateTimeStyles](#) value with the parsing method.

The format provider is also used to interpret an ambiguous numeric date. It's unclear which components of the date represented by the string "02/03/04" are the month, day, and year. The components are interpreted according to the order of similar date formats in the format provider.

Parse

The following example illustrates the use of the [DateTime.Parse](#) method to convert a `string` into a [DateTime](#). This example uses the culture associated with the current thread. If the [CultureInfo](#) associated with the current culture can't parse the input string, a [FormatException](#) is thrown.

Tip

All the C# samples in this article run in your browser. Press the **Run** button to see the output. You can also edit them to experiment yourself.

Note

These examples are available in the GitHub docs repo for both [C#](#) and [Visual Basic](#).

C#

```
string dateInput = "Jan 1, 2009";
var parsedDate = DateTime.Parse(dateInput);
Console.WriteLine(parsedDate);
// Displays the following output on a system whose culture is en-US:
//      1/1/2009 00:00:00
```

You can also explicitly define the culture whose formatting conventions are used when you parse a string. You specify one of the standard [DateTimeFormatInfo](#) objects returned by the [CultureInfo.DateTimeFormat](#) property. The following example uses a format provider to parse a German string into a [DateTime](#). It creates a [CultureInfo](#) representing the `de-DE` culture. That [CultureInfo](#) object ensures successful parsing of this particular string. This process precludes whatever setting is in the [CurrentCulture](#) of the [CurrentThread](#).

C#

```
var cultureInfo = new CultureInfo("de-DE");
string dateString = "12 Juni 2008";
var dateTime = DateTime.Parse(dateString, cultureInfo);
Console.WriteLine(dateTime);
// The example displays the following output:
//      6/12/2008 00:00:00
```

However, you can use overloads of the [Parse](#) method to specify custom format providers. The [Parse](#) method doesn't support parsing non-standard formats. To parse a date and time expressed in a non-standard format, use the [ParseExact](#) method instead.

The following example uses the [DateTimeStyles](#) enumeration to specify that the current date and time information shouldn't be added to the [DateTime](#) for unspecified fields.

C#

```
var cultureInfo = new CultureInfo("de-DE");
string dateString = "12 Juni 2008";
var dateTime = DateTime.Parse(dateString, cultureInfo,
                             DateTimeStyles.NoCurrentDateDefault);
Console.WriteLine(dateTime);
// The example displays the following output if the current culture is en-US:
//      6/12/2008 00:00:00
```

ParseExact

The [DateTime.ParseExact](#) method converts a string to a [DateTime](#) object if it conforms to one of the specified string patterns. When a string that isn't one of the forms specified is passed to this method, a [FormatException](#) is thrown. You can specify one of the standard date and time format specifiers or a combination of the custom format specifiers. Using the custom format specifiers, it's possible for you to construct a custom recognition string. For an explanation of the specifiers, see the articles on [standard date and time format strings](#) and [custom date and time format strings](#).

In the following example, the [DateTime.ParseExact](#) method is passed a string object to parse, followed by a format specifier, followed by a [CultureInfo](#) object. This [ParseExact](#) method can only parse strings that follow the long date pattern in the `en-US` culture.

C#

```
var cultureInfo = new CultureInfo("en-US");
string[] dateStrings = { " Friday, April 10, 2009", "Friday, April 10, 2009"
};
foreach (string dateString in dateStrings)
{
    try
    {
        var dateTime = DateTime.ParseExact(dateString, "D", cultureInfo);
        Console.WriteLine(dateTime);
    }
    catch (FormatException)
    {
        Console.WriteLine("Unable to parse '{0}'", dateString);
    }
}
// The example displays the following output:
//      Unable to parse ' Friday, April 10, 2009'
//      4/10/2009 00:00:00
```

Each overload of the [Parse](#) and [ParseExact](#) methods also has an [IFormatProvider](#) parameter that provides culture-specific information about the formatting of the string. The [IFormatProvider](#) object is a [CultureInfo](#) object that represents a standard culture or a [DateTimeFormatInfo](#) object that is returned by the [CultureInfo.DateTimeFormat](#) property. [ParseExact](#) also uses an additional string or string array argument that defines one or more custom date and time formats.

See also

- [Parsing strings](#)

- [Formatting types](#)
- [Type conversion in .NET](#)
- [Standard date and time formats](#)
- [Custom date and time format strings](#)

 [Collaborate with us on GitHub](#)

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

Parsing Other Strings in .NET

Article • 03/11/2022

In addition to numeric and [DateTime](#) strings, you can also parse strings that represent the types [Char](#), [Boolean](#), and [Enum](#) into data types.

Char

The static `parse` method associated with the **Char** data type is useful for converting a string that contains a single character into its Unicode value. The following code example parses a string into a Unicode character.

C#

```
string MyString1 = "A";
char MyChar = Char.Parse(MyString1);
// MyChar now contains a Unicode "A" character.
```

Boolean

The **Boolean** data type contains a `Parse` method that you can use to convert a string that represents a Boolean value into an actual **Boolean** type. This method is not case-sensitive and can successfully parse a string containing "True" or "False." The `Parse` method associated with the **Boolean** type can also parse strings that are surrounded by white spaces. If any other string is passed, a [FormatException](#) is thrown.

The following code example uses the `Parse` method to convert a string into a Boolean value.

C#

```
string MyString2 = "True";
bool MyBool = bool.Parse(MyString2);
// MyBool now contains a True Boolean value.
```

Enumeration

You can use the static `Parse` method to initialize an enumeration type to the value of a string. This method accepts the enumeration type you are parsing, the string to parse, and an optional Boolean flag indicating whether or not the parse is case-sensitive. The

string you are parsing can contain several values separated by commas, which can be preceded or followed by one or more empty spaces (also called white spaces). When the string contains multiple values, the value of the returned object is the value of all specified values combined with a bitwise OR operation.

The following example uses the `Parse` method to convert a string representation into an enumeration value. The `DayOfWeek` enumeration is initialized to `Thursday` from a string.

C#

```
string MyString3 = "Thursday";
DayOfWeek MyDays = (DayOfWeek)Enum.Parse(typeof(DayOfWeek), MyString3);
Console.WriteLine(MyDays);
// The result is Thursday.
```

See also

- [Parsing Strings](#)
- [Formatting Types](#)
- [Type Conversion in .NET](#)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

[Open a documentation issue](#)

[Provide product feedback](#)

System.String class

Article • 01/08/2024

This article provides supplementary remarks to the reference documentation for this API.

A string is a sequential collection of characters that's used to represent text. A [String](#) object is a sequential collection of [System.Char](#) objects that represent a string; a [System.Char](#) object corresponds to a UTF-16 code unit. The value of the [String](#) object is the content of the sequential collection of [System.Char](#) objects, and that value is immutable (that is, it is read-only). For more information about the immutability of strings, see the [Immutability and the StringBuilder class](#) section. The maximum size of a [String](#) object in memory is 2-GB, or about 1 billion characters.

For more information about Unicode, UTF-16, code units, code points, and the [Char](#) and [Rune](#) types, see [Introduction to character encoding in .NET](#).

Instantiate a String object

You can instantiate a [String](#) object in the following ways:

- By assigning a string literal to a [String](#) variable. This is the most commonly used method for creating a string. The following example uses assignment to create several strings. Note that in C# and F#, because the backslash (\) is an escape character, literal backslashes in a string must be escaped or the entire string must be @-quoted.

C#

```
string string1 = "This is a string created by assignment.";
Console.WriteLine(string1);
string string2a = "The path is C:\\PublicDocuments\\Report1.doc";
Console.WriteLine(string2a);
string string2b = @"/The path is C:\\PublicDocuments\\Report1.doc";
Console.WriteLine(string2b);
// The example displays the following output:
//      This is a string created by assignment.
//      The path is C:\\PublicDocuments\\Report1.doc
//      The path is C:\\PublicDocuments\\Report1.doc
```

- By calling a [String](#) class constructor. The following example instantiates strings by calling several class constructors. Note that some of the constructors include pointers to character arrays or signed byte arrays as parameters. Visual Basic does

not support calls to these constructors. For detailed information about [String](#) constructors, see the [String constructor summary](#).

```
C#  
  
char[] chars = { 'w', 'o', 'r', 'd' };  
sbyte[] bytes = { 0x41, 0x42, 0x43, 0x44, 0x45, 0x00 };  
  
// Create a string from a character array.  
string string1 = new string(chars);  
Console.WriteLine(string1);  
  
// Create a string that consists of a character repeated 20 times.  
string string2 = new string('c', 20);  
Console.WriteLine(string2);  
  
string stringFromBytes = null;  
string stringFromChars = null;  
unsafe  
{  
    fixed (sbyte* pbytes = bytes)  
    {  
        // Create a string from a pointer to a signed byte array.  
        stringFromBytes = new string(pbytes);  
    }  
    fixed (char* pchars = chars)  
    {  
        // Create a string from a pointer to a character array.  
        stringFromChars = new string(pchars);  
    }  
}  
Console.WriteLine(stringFromBytes);  
Console.WriteLine(stringFromChars);  
// The example displays the following output:  
//      word  
//      ccccccccccccccccccc  
//      ABCDE  
//      word
```

- By using the string concatenation operator (+ in C# and F#, and & or + in Visual Basic) to create a single string from any combination of [String](#) instances and string literals. The following example illustrates the use of the string concatenation operator.

```
C#  
  
string string1 = "Today is " + DateTime.Now.ToString("D") + ".";  
Console.WriteLine(string1);  
  
string string2 = "This is one sentence. " + "This is a second. ";  
string2 += "This is a third sentence.";
```

```
Console.WriteLine(string2);
// The example displays output like the following:
//     Today is Tuesday, July 06, 2011.
//     This is one sentence. This is a second. This is a third sentence.
```

- By retrieving a property or calling a method that returns a string. The following example uses the methods of the [String](#) class to extract a substring from a larger string.

C#

```
string sentence = "This sentence has five words.";
// Extract the second word.
int startPosition = sentence.IndexOf(" ") + 1;
string word2 = sentence.Substring(startPosition,
                                  sentence.IndexOf(" ", startPosition)
                                  - startPosition);
Console.WriteLine("Second word: " + word2);
// The example displays the following output:
//     Second word: sentence
```

- By calling a formatting method to convert a value or object to its string representation. The following example uses the [composite formatting](#) feature to embed the string representation of two objects into a string.

C#

```
DateTime dateAndTime = new DateTime(2011, 7, 6, 7, 32, 0);
double temperature = 68.3;
string result = String.Format("At {0:t} on {0:D}, the temperature was
{1:F1} degrees Fahrenheit.", 
                               dateAndTime, temperature);
Console.WriteLine(result);
// The example displays the following output:
//     At 7:32 AM on Wednesday, July 06, 2011, the temperature was
//     68.3 degrees Fahrenheit.
```

Char objects and Unicode characters

Each character in a string is defined by a Unicode scalar value, also called a Unicode code point or the ordinal (numeric) value of the Unicode character. Each code point is encoded by using UTF-16 encoding, and the numeric value of each element of the encoding is represented by a [Char](#) object.

 Note

Note that, because a `String` instance consists of a sequential collection of UTF-16 code units, it is possible to create a `String` object that is not a well-formed Unicode string. For example, it is possible to create a string that has a low surrogate without a corresponding high surrogate. Although some methods, such as the methods of encoding and decoding objects in the `System.Text` namespace, may perform checks to ensure that strings are well-formed, `String` class members don't ensure that a string is well-formed.

A single `Char` object usually represents a single code point; that is, the numeric value of the `Char` equals the code point. For example, the code point for the character "a" is U+0061. However, a code point might require more than one encoded element (more than one `Char` object). The Unicode standard defines two types of characters that correspond to multiple `Char` objects: graphemes, and Unicode supplementary code points that correspond to characters in the Unicode supplementary planes.

- A grapheme is represented by a base character followed by one or more combining characters. For example, the character ä is represented by a `Char` object whose code point is U+0061 followed by a `Char` object whose code point is U+0308. This character can also be defined by a single `Char` object that has a code point of U+00E4. As the following example shows, a culture-sensitive comparison for equality indicates that these two representations are equal, although an ordinary ordinal comparison does not. However, if the two strings are normalized, an ordinal comparison also indicates that they are equal. (For more information on normalizing strings, see the [Normalization](#) section.)

C#

```
using System;
using System.Globalization;
using System.IO;

public class Example5
{
    public static void Main()
    {
        StreamWriter sw = new StreamWriter(@".\graphemes.txt");
        string grapheme = "\u0061\u0308";
        sw.WriteLine(grapheme);

        string singleChar = "\u00e4";
        sw.WriteLine(singleChar);

        sw.WriteLine("{0} = {1} (Culture-sensitive): {2}", grapheme,
singleChar,
            String.Equals(grapheme, singleChar,
StringComparison.CurrentCulture));
```

```

        sw.WriteLine("{0} = {1} (Ordinal): {2}", grapheme, singleChar,
                    String.Equals(grapheme, singleChar,
                                  StringComparison.OrdinalIgnoreCase));
        sw.WriteLine("{0} = {1} (Normalized Ordinal): {2}", grapheme,
                    singleChar,
                    String.Equals(grapheme.Normalize(),
                                  singleChar.Normalize(),
                                  StringComparison.OrdinalIgnoreCase));
    sw.Close();
}
}

// The example produces the following output:
//      ä
//      ä
//      ä = ä (Culture-sensitive): True
//      ä = ä (Ordinal): False
//      ä = ä (Normalized Ordinal): True

```

- A Unicode supplementary code point (a surrogate pair) is represented by a [Char](#) object whose code point is a high surrogate followed by a [Char](#) object whose code point is a low surrogate. The code units of high surrogates range from U+D800 to U+DBFF. The code units of low surrogates range from U+DC00 to U+DFFF. Surrogate pairs are used to represent characters in the 16 Unicode supplementary planes. The following example creates a surrogate character and passes it to the [Char.IsSurrogatePair\(Char, Char\)](#) method to determine whether it is a surrogate pair.

C#

```

string surrogate = "\uD800\uDC03";
for (int ctr = 0; ctr < surrogate.Length; ctr++)
    Console.Write($"U+{(ushort)surrogate[ctr]:X2} ");

Console.WriteLine();
Console.WriteLine("  Is Surrogate Pair: {0}",
                Char.IsSurrogatePair(surrogate[0], surrogate[1]));
// The example displays the following output:
//      U+D800 U+DC03
//      Is Surrogate Pair: True

```

The Unicode standard

Characters in a string are represented by UTF-16 encoded code units, which correspond to [Char](#) values.

Each character in a string has an associated Unicode character category, which is represented in .NET by the [UnicodeCategory](#) enumeration. The category of a character

or a surrogate pair can be determined by calling the [CharUnicodeInfo.GetUnicodeCategory](#) method.

.NET maintains its own table of characters and their corresponding categories, which ensures that a specific version of a .NET implementation running on different platforms returns identical character category information. On all .NET versions and across all OS platforms, character category information is provided by the [Unicode Character Database](#).

The following table lists .NET versions and the versions of the Unicode Standard on which their character categories are based.

[\[+\] Expand table](#)

| .NET version | Version of the Unicode Standard |
|---|--|
| .NET Framework 1.1 | The Unicode Standard, Version 4.0.0 |
| .NET Framework 2.0 | The Unicode Standard, Version 5.0.0 |
| .NET Framework 3.5 | The Unicode Standard, Version 5.0.0 |
| .NET Framework 4 | The Unicode Standard, Version 5.0.0 |
| .NET Framework 4.5 | The Unicode Standard, Version 6.3.0 |
| .NET Framework 4.5.1 | The Unicode Standard, Version 6.3.0 |
| .NET Framework 4.5.2 | The Unicode Standard, Version 6.3.0 |
| .NET Framework 4.6 | The Unicode Standard, Version 6.3.0 |
| .NET Framework 4.6.1 | The Unicode Standard, Version 6.3.0 |
| .NET Framework 4.6.2 and later versions | The Unicode Standard, Version 8.0.0 |
| .NET Core 2.1 | The Unicode Standard, Version 8.0.0 |
| .NET Core 3.1 | The Unicode Standard, Version 11.0.0 |
| .NET 5 | The Unicode Standard, Version 13.0.0 |

In addition, .NET supports string comparison and sorting based on the Unicode standard. .NET Framework 4 and previous versions maintain their own table of string data. This is also true of versions of .NET Framework starting with .NET Framework 4.5 running on Windows 7. Starting with .NET Framework 4.5 running on Windows 8 and later versions of the Windows operating system, the runtime delegates string comparison and sorting operations to the operating system. On .NET Core and .NET 5+,

String comparison and sorting information is provided by [International Components for Unicode](#) libraries (except on Windows versions prior to Windows 10 May 2019 Update). The following table lists the versions of .NET and the versions of the Unicode Standard on which character comparison and sorting are based.

 [Expand table](#)

| .NET version | Version of the Unicode Standard |
|---|--|
| .NET Framework 1.1 | The Unicode Standard, Version 4.0.0 |
| .NET Framework 2.0 | The Unicode Standard, Version 5.0.0 |
| .NET Framework 3.5 | The Unicode Standard, Version 5.0.0 |
| .NET Framework 4 | The Unicode Standard, Version 5.0.0 |
| .NET Framework 4.5 and later on Windows 7 | The Unicode Standard, Version 5.0.0 |
| .NET Framework 4.5 and later on Windows 8 and later Windows operating systems | The Unicode Standard, Version 6.3.0 |
| .NET Core and .NET 5+ | Depends on the version of the Unicode Standard supported by the underlying operating system. |

Embedded null characters

In .NET, a [String](#) object can include embedded null characters, which count as a part of the string's length. However, in some languages such as C and C++, a null character indicates the end of a string; it is not considered a part of the string and is not counted as part of the string's length. This means that the following common assumptions that C and C++ programmers or libraries written in C or C++ might make about strings are not necessarily valid when applied to [String](#) objects:

- The value returned by the `strlen` or `wcslen` functions does not necessarily equal [String.Length](#).
- The string created by the `strcpy_s` or `wcsncpy_s` functions is not necessarily identical to the string created by the [String.Copy](#) method.

You should ensure that native C and C++ code that instantiates [String](#) objects, and code that is passed [String](#) objects through platform invoke, don't assume that an embedded null character marks the end of the string.

Embedded null characters in a string are also treated differently when a string is sorted (or compared) and when a string is searched. Null characters are ignored when performing culture-sensitive comparisons between two strings, including comparisons using the invariant culture. They are considered only for ordinal or case-insensitive ordinal comparisons. On the other hand, embedded null characters are always considered when searching a string with methods such as [Contains](#), [StartsWith](#), and [IndexOf](#).

Strings and indexes

An index is the position of a [Char](#) object (not a Unicode character) in a [String](#). An index is a zero-based, nonnegative number that starts from the first position in the string, which is index position zero. A number of search methods, such as [IndexOf](#) and [LastIndexOf](#), return the index of a character or substring in the string instance.

The [Chars\[\]](#) property lets you access individual [Char](#) objects by their index position in the string. Because the [Chars\[\]](#) property is the default property (in Visual Basic) or the indexer (in C# and F#), you can access the individual [Char](#) objects in a string by using code such as the following. This code looks for white space or punctuation characters in a string to determine how many words the string contains.

C#

```
string s1 = "This string consists of a single short sentence.";
int nWords = 0;

s1 = s1.Trim();
for (int ctr = 0; ctr < s1.Length; ctr++) {
    if (Char.IsPunctuation(s1[ctr]) | Char.IsWhiteSpace(s1[ctr]))
        nWords++;
}
Console.WriteLine("The sentence\n  {0}\nhas {1} words.",
                  s1, nWords);
// The example displays the following output:
//      The sentence
//      This string consists of a single short sentence.
//      has 8 words.
```

Because the [String](#) class implements the [IEnumerable](#) interface, you can also iterate through the [Char](#) objects in a string by using a [foreach](#) construct, as the following example shows.

C#

```

string s1 = "This string consists of a single short sentence.";
int nWords = 0;

s1 = s1.Trim();
foreach (var ch in s1) {
    if (Char.IsPunctuation(ch) | Char.IsWhiteSpace(ch))
        nWords++;
}
Console.WriteLine("The sentence\n  {0}\nhas {1} words.",
                  s1, nWords);
// The example displays the following output:
//      The sentence
//      This string consists of a single short sentence.
//      has 8 words.

```

Consecutive index values might not correspond to consecutive Unicode characters, because a Unicode character might be encoded as more than one [Char](#) object. In particular, a string may contain multi-character units of text that are formed by a base character followed by one or more combining characters or by surrogate pairs. To work with Unicode characters instead of [Char](#) objects, use the [System.Globalization.StringInfo](#) and [TextElementEnumerator](#) classes, or the [String.EnumerateRunes](#) method and the [Rune](#) struct. The following example illustrates the difference between code that works with [Char](#) objects and code that works with Unicode characters. It compares the number of characters or text elements in each word of a sentence. The string includes two sequences of a base character followed by a combining character.

C#

```

// First sentence of The Mystery of the Yellow Room, by Leroux.
string opening = "Ce n'est pas sans une certaine émotion que "+
                 "je commence à raconter ici les aventures " +
                 "extraordinaires de Joseph Rouletabille.';

// Character counters.
int nChars = 0;
// Objects to store word count.
List<int> chars = new List<int>();
List<int> elements = new List<int>();

foreach (var ch in opening) {
    // Skip the ' character.
    if (ch == '\u0027') continue;

    if (Char.IsWhiteSpace(ch) | (Char.IsPunctuation(ch))) {
        chars.Add(nChars);
        nChars = 0;
    }
    else {
        nChars++;
    }
}

```

```

}

System.Globalization.TextElementEnumerator te =
    System.Globalization.StringInfo.GetTextElementEnumerator(opening);
while (te.MoveNext()) {
    string s = te.GetTextElement();
    // Skip the ' character.
    if (s == "\u0027") continue;
    if (String.IsNullOrEmpty(s.Trim()) | (s.Length == 1 &&
Char.IsPunctuation(Convert.ToChar(s)))) {
        elements.Add(nChars);
        nChars = 0;
    }
    else {
        nChars++;
    }
}

// Display character counts.
Console.WriteLine("{0,6} {1,20} {2,20}",
    "Word #", "Char Objects", "Characters");
for (int ctr = 0; ctr < chars.Count; ctr++)
    Console.WriteLine("{0,6} {1,20} {2,20}",
        ctr, chars[ctr], elements[ctr]);
// The example displays the following output:
//      Word #      Char Objects      Characters
//          0                  2                  2
//          1                  4                  4
//          2                  3                  3
//          3                  4                  4
//          4                  3                  3
//          5                  8                  8
//          6                  8                  7
//          7                  3                  3
//          8                  2                  2
//          9                  8                  8
//         10                  2                  1
//         11                  8                  8
//         12                  3                  3
//         13                  3                  3
//         14                  9                  9
//         15                 15                 15
//         16                  2                  2
//         17                  6                  6
//         18                 12                 12

```

This example works with text elements by using the [StringInfo.GetTextElementEnumerator](#) method and the [TextElementEnumerator](#) class to enumerate all the text elements in a string. You can also retrieve an array that contains the starting index of each text element by calling the [StringInfo.ParseCombiningCharacters](#) method.

For more information about working with units of text rather than individual `Char` values, see [Introduction to character encoding in .NET](#).

Null strings and empty strings

A string that has been declared but has not been assigned a value is `null`. Attempting to call methods on that string throws a `NullReferenceException`. A null string is different from an empty string, which is a string whose value is `""` or `String.Empty`. In some cases, passing either a null string or an empty string as an argument in a method call throws an exception. For example, passing a null string to the `Int32.Parse` method throws an `ArgumentNullException`, and passing an empty string throws a `FormatException`. In other cases, a method argument can be either a null string or an empty string. For example, if you are providing an `IFormattable` implementation for a class, you want to equate both a null string and an empty string with the general ("G") format specifier.

The `String` class includes the following two convenience methods that enable you to test whether a string is `null` or empty:

- `IsNullOrEmpty`, which indicates whether a string is either `null` or is equal to `String.Empty`. This method eliminates the need to use code such as the following:

```
C#  
  
if (str == null || str.Equals(String.Empty))
```

- `IsNullOrWhiteSpace`, which indicates whether a string is `null`, equals `String.Empty`, or consists exclusively of white-space characters. This method eliminates the need to use code such as the following:

```
C#  
  
if (str == null || str.Equals(String.Empty) ||  
str.Trim().Equals(String.Empty))
```

The following example uses the `IsNullOrEmpty` method in the `IFormattable.ToString` implementation of a custom `Temperature` class. The method supports the "G", "C", "F", and "K" format strings. If an empty format string or a format string whose value is `null` is passed to the method, its value is changed to the "G" format string.

```
C#
```

```

public string ToString(string format, IFormatProvider provider)
{
    if (String.IsNullOrEmpty(format)) format = "G";
    if (provider == null) provider = CultureInfo.CurrentCulture;

    switch (format.ToUpperInvariant())
    {
        // Return degrees in Celsius.
        case "G":
        case "C":
            return temp.ToString("F2", provider) + "°C";
        // Return degrees in Fahrenheit.
        case "F":
            return (temp * 9 / 5 + 32).ToString("F2", provider) + "°F";
        // Return degrees in Kelvin.
        case "K":
            return (temp + 273.15).ToString();
        default:
            throw new FormatException(
                String.Format("The {0} format string is not supported.", 
                format));
    }
}

```

Immutability and the StringBuilder class

A [String](#) object is called immutable (read-only), because its value cannot be modified after it has been created. Methods that appear to modify a [String](#) object actually return a new [String](#) object that contains the modification.

Because strings are immutable, string manipulation routines that perform repeated additions or deletions to what appears to be a single string can exact a significant performance penalty. For example, the following code uses a random number generator to create a string with 1000 characters in the range 0x0001 to 0x052F. Although the code appears to use string concatenation to append a new character to the existing string named `str`, it actually creates a new [String](#) object for each concatenation operation.

C#

```

using System;
using System.IO;
using System.Text;

public class Example6
{
    public static void Main()
    {

```

```

Random rnd = new Random();

string str = String.Empty;
StreamWriter sw = new StreamWriter(@".\StringFile.txt",
                                false, Encoding.Unicode);

for (int ctr = 0; ctr <= 1000; ctr++) {
    str += (char)rnd.Next(1, 0x0530);
    if (str.Length % 60 == 0)
        str += Environment.NewLine;
}
sw.Write(str);
sw.Close();
}
}

```

You can use the [StringBuilder](#) class instead of the [String](#) class for operations that make multiple changes to the value of a string. Unlike instances of the [String](#) class, [StringBuilder](#) objects are mutable; when you concatenate, append, or delete substrings from a string, the operations are performed on a single string. When you have finished modifying the value of a [StringBuilder](#) object, you can call its [StringBuilder.ToString](#) method to convert it to a string. The following example replaces the [String](#) used in the previous example to concatenate 1000 random characters in the range to 0x0001 to 0x052F with a [StringBuilder](#) object.

C#

```

using System;
using System.IO;
using System.Text;

public class Example10
{
    public static void Main()
    {
        Random rnd = new Random();
        StringBuilder sb = new StringBuilder();
        StreamWriter sw = new StreamWriter(@".\StringFile.txt",
                                         false, Encoding.Unicode);

        for (int ctr = 0; ctr <= 1000; ctr++) {
            sb.Append((char)rnd.Next(1, 0x0530));
            if (sb.Length % 60 == 0)
                sb.AppendLine();
        }
        sw.Write(sb.ToString());
        sw.Close();
    }
}

```

Ordinal vs. culture-sensitive operations

Members of the [String](#) class perform either ordinal or culture-sensitive (linguistic) operations on a [String](#) object. An ordinal operation acts on the numeric value of each [Char](#) object. A culture-sensitive operation acts on the value of the [String](#) object, and takes culture-specific casing, sorting, formatting, and parsing rules into account. Culture-sensitive operations execute in the context of an explicitly declared culture or the implicit current culture. The two kinds of operations can produce very different results when they are performed on the same string.

.NET also supports culture-insensitive linguistic string operations by using the invariant culture ([CultureInfo.InvariantCulture](#)), which is loosely based on the culture settings of the English language independent of region. Unlike other [System.Globalization.CultureInfo](#) settings, the settings of the invariant culture are guaranteed to remain consistent on a single computer, from system to system, and across versions of .NET. The invariant culture can be seen as a kind of black box that ensures stability of string comparisons and ordering across all cultures.

Important

If your application makes a security decision about a symbolic identifier such as a file name or named pipe, or about persisted data such as the text-based data in an XML file, the operation should use an ordinal comparison instead of a culture-sensitive comparison. This is because a culture-sensitive comparison can yield different results depending on the culture in effect, whereas an ordinal comparison depends solely on the binary value of the compared characters.

Important

Most methods that perform string operations include an overload that has a parameter of type [StringComparison](#), which enables you to specify whether the method performs an ordinal or culture-sensitive operation. In general, you should call this overload to make the intent of your method call clear. For best practices and guidance for using ordinal and culture-sensitive operations on strings, see [Best Practices for Using Strings](#).

Operations for casing, parsing and formatting, comparison and sorting, and testing for equality can be either ordinal or culture-sensitive. The following sections discuss each category of operation.

💡 Tip

You should always call a method overload that makes the intent of your method call clear. For example, instead of calling the `Compare(String, String)` method to perform a culture-sensitive comparison of two strings by using the conventions of the current culture, you should call the `Compare(String, String, StringComparison)` method with a value of `StringComparison.CurrentCulture` for the `comparisonType` argument. For more information, see [Best Practices for Using Strings](#).

You can download the sorting weight tables, a set of text files that contain information on the character weights used in sorting and comparison operations, from the following links:

- Windows (.NET Framework and .NET Core): [Sorting Weight Tables ↗](#)
- Windows 10 May 2019 Update or later (.NET 5+) and Linux and macOS (.NET Core and .NET 5+): [Default Unicode Collation Element Table ↗](#)

Casing

Casing rules determine how to change the capitalization of a Unicode character; for example, from lowercase to uppercase. Often, a casing operation is performed before a string comparison. For example, a string might be converted to uppercase so that it can be compared with another uppercase string. You can convert the characters in a string to lowercase by calling the `ToLower` or `ToLowerInvariant` method, and you can convert them to uppercase by calling the `ToUpper` or `ToUpperInvariant` method. In addition, you can use the `TextInfo.ToTitleCase` method to convert a string to title case.

ⓘ Note

.NET Core running on Linux and macOS systems only: The collation behavior for the C and Posix cultures is always case-sensitive because these cultures do not use the expected Unicode collation order. We recommend that you use a culture other than C or Posix for performing culture-sensitive, case-insensitive sorting operations.

Casing operations can be based on the rules of the current culture, a specified culture, or the invariant culture. Because case mappings can vary depending on the culture used, the result of casing operations can vary based on culture. The actual differences in casing are of three kinds:

- Differences in the case mapping of LATIN CAPITAL LETTER I (U+0049), LATIN SMALL LETTER I (U+0069), LATIN CAPITAL LETTER I WITH DOT ABOVE (U+0130), and LATIN SMALL LETTER DOTLESS I (U+0131). In the tr-TR (Turkish (Turkey)) and az-Latn-AZ (Azerbaijan, Latin) cultures, and in the tr, az, and az-Latn neutral cultures, the lowercase equivalent of LATIN CAPITAL LETTER I is LATIN SMALL LETTER DOTLESS I, and the uppercase equivalent of LATIN SMALL LETTER I is LATIN CAPITAL LETTER I WITH DOT ABOVE. In all other cultures, including the invariant culture, LATIN SMALL LETTER I and LATIN CAPITAL LETTER I are lowercase and uppercase equivalents.

The following example demonstrates how a string comparison designed to prevent file system access can fail if it relies on a culture-sensitive casing comparison. (The casing conventions of the invariant culture should have been used.)

C#

```
using System;
using System.Globalization;
using System.Threading;

public class Example1
{
    const string disallowed = "file";

    public static void Main()
    {

        IsAccessAllowed(@"FILE:\\\\c:\\users\\user001\\documents\\FinancialInfo.txt");
    }

    private static void IsAccessAllowed(String resource)
    {
        CultureInfo[] cultures = { CultureInfo.CreateSpecificCulture("en-US"),
                                    CultureInfo.CreateSpecificCulture("tr-TR") };
        String scheme = null;
        int index = resource.IndexOfAny( new Char[] { '\\', '/' } );
        if (index > 0)
            scheme = resource.Substring(0, index - 1);

        // Change the current culture and perform the comparison.
        foreach (var culture in cultures) {
            Thread.CurrentThread.CurrentCulture = culture;
            Console.WriteLine("Culture: {0}",
                CultureInfo.CurrentCulture.DisplayName);
            Console.WriteLine(resource);
            Console.WriteLine("Access allowed: {0}",
                ! String.Equals(disallowed, scheme,

```

```

        StringComparison.CurrentCultureIgnoreCase));
        Console.WriteLine();
    }
}
// The example displays the following output:
//      Culture: English (United States)
//      FILE:\\\\c:\\users\\user001\\documents\\FinancialInfo.txt
//      Access allowed: False
//
//      Culture: Turkish (Turkey)
//      FILE:\\\\c:\\users\\user001\\documents\\FinancialInfo.txt
//      Access allowed: True

```

- Differences in case mappings between the invariant culture and all other cultures. In these cases, using the casing rules of the invariant culture to change a character to uppercase or lowercase returns the same character. For all other cultures, it returns a different character. Some of the affected characters are listed in the following table.

[\[+\] Expand table](#)

| Character | If changed to | Returns |
|--|---------------|---|
| MICRON SIGN (U+00B5) | Uppercase | GREEK CAPITAL LETTER MU (U+-39C) |
| LATIN CAPITAL LETTER I WITH DOT ABOVE (U+0130) | Lowercase | LATIN SMALL LETTER I (U+0069) |
| LATIN SMALL LETTER DOTLESS I (U+0131) | Uppercase | LATIN CAPITAL LETTER I (U+0049) |
| LATIN SMALL LETTER LONG S (U+017F) | Uppercase | LATIN CAPITAL LETTER S (U+0053) |
| LATIN CAPITAL LETTER D WITH SMALL LETTER Z WITH CARON (U+01C5) | Lowercase | LATIN SMALL LETTER DZ WITH CARON (U+01C6) |
| COMBINING GREEK YPOGEGRAMMENI (U+0345) | Uppercase | GREEK CAPITAL LETTER IOTA (U+0399) |

- Differences in case mappings of two-letter mixed-case pairs in the ASCII character range. In most cultures, a two-letter mixed-case pair is equal to the equivalent two-letter uppercase or lowercase pair. This is not true for the following two-letter pairs in the following cultures, because in each case they are compared to a digraph:

- "IJ" and "nJ" in the hr-HR (Croatian (Croatia)) culture.
- "cH" in the cs-CZ (Czech (Czech Republic)) and sk-SK (Slovak (Slovakia)) cultures.
- "aA" in the da-DK (Danish (Denmark)) culture.
- "cS", "dZ", "dZS", "nY", "sZ", "tY", and "zS" in the hu-HU (Hungarian (Hungary)) culture.
- "cH" and "IL" in the es-ES_tradnl (Spanish (Spain, Traditional Sort)) culture.
- "cH", "gl", "kH", "nG" "nH", "pH", "qU", "tH", and "tR" in the vi-VN (Vietnamese (Vietnam)) culture.

However, it is unusual to encounter a situation in which a culture-sensitive comparison of these pairs creates problems, because these pairs are uncommon in fixed strings or identifiers.

The following example illustrates some of the differences in casing rules between cultures when converting strings to uppercase.

C#

```
using System;
using System.Globalization;
using System.IO;

public class Example
{
    public static void Main()
    {
        StreamWriter sw = new StreamWriter(@".\case.txt");
        string[] words = { "file", "sifir", "ßenana" };
        CultureInfo[] cultures = { CultureInfo.InvariantCulture,
                                   new CultureInfo("en-US"),
                                   new CultureInfo("tr-TR") };

        foreach (var word in words) {
            sw.WriteLine("{0}:", word);
            foreach (var culture in cultures) {
                string name = String.IsNullOrEmpty(culture.Name) ?
                    "Invariant" : culture.Name;
                string upperWord = word.ToUpper(culture);
                sw.WriteLine("  {0,10}: {1,7} {2, 38}", name,
                           upperWord, ShowHexValue(upperWord));
            }
            sw.WriteLine();
        }
        sw.Close();
    }

    private static string ShowHexValue(string s)
    {
        string retval = null;
        foreach (var ch in s) {
```

```

        byte[] bytes = BitConverter.GetBytes(ch);
        retval += String.Format("{0:X2} {1:X2} ", bytes[1], bytes[0]);
    }
    return retval;
}
// The example displays the following output:
//   file:
//       Invariant: FILE          00 46 00 49 00 4C 00 45
//       en-US:     FILE          00 46 00 49 00 4C 00 45
//       tr-TR:     FILE          00 46 01 30 00 4C 00 45
//
//   sıfır:
//       Invariant: SıFıR        00 53 01 31 00 46 01 31 00 52
//       en-US:     SIFIR         00 53 00 49 00 46 00 49 00 52
//       tr-TR:     SIFIR         00 53 00 49 00 46 00 49 00 52
//
//   Đenana:
//       Invariant: ĐENANA      01 C5 00 45 00 4E 00 41 00 4E 00 41
//       en-US:     ĐENANA      01 C4 00 45 00 4E 00 41 00 4E 00 41
//       tr-TR:     ĐENANA      01 C4 00 45 00 4E 00 41 00 4E 00 41

```

Parsing and formatting

Formatting and parsing are inverse operations. Formatting rules determine how to convert a value, such as a date and time or a number, to its string representation, whereas parsing rules determine how to convert a string representation to a value such as a date and time. Both formatting and parsing rules are dependent on cultural conventions. The following example illustrates the ambiguity that can arise when interpreting a culture-specific date string. Without knowing the conventions of the culture that was used to produce a date string, it is not possible to know whether 03/01/2011, 3/1/2011, and 01/03/2011 represent January 3, 2011 or March 1, 2011.

C#

```

using System;
using System.Globalization;

public class Example9
{
    public static void Main()
    {
        DateTime date = new DateTime(2011, 3, 1);
        CultureInfo[] cultures = { CultureInfo.InvariantCulture,
                                  new CultureInfo("en-US"),
                                  new CultureInfo("fr-FR") };

        foreach (var culture in cultures)
            Console.WriteLine("{0,-12} {1}", String.IsNullOrEmpty(culture.Name)

```

```

    ?
        "Invariant" : culture.Name,
        date.ToString("d", culture));
    }
}
// The example displays the following output:
//      Invariant      03/01/2011
//      en-US          3/1/2011
//      fr-FR          01/03/2011

```

Similarly, as the following example shows, a single string can produce different dates depending on the culture whose conventions are used in the parsing operation.

C#

```

using System;
using System.Globalization;

public class Example15
{
    public static void Main()
    {
        string dateString = "07/10/2011";
        CultureInfo[] cultures = { CultureInfo.InvariantCulture,
                                   CultureInfo.CreateSpecificCulture("en-GB"),
                                   CultureInfo.CreateSpecificCulture("en-US") };
        Console.WriteLine("{0,-12} {1,10} {2,8} {3,8}\n", "Date String",
"Culture",
                           "Month", "Day");
        foreach (var culture in cultures) {
            DateTime date = DateTime.Parse(dateString, culture);
            Console.WriteLine("{0,-12} {1,10} {2,8} {3,8}", dateString,
                           String.IsNullOrEmpty(culture.Name) ?
                           "Invariant" : culture.Name,
                           date.Month, date.Day);
        }
    }
}
// The example displays the following output:
//      Date String      Culture      Month      Day
//
//      07/10/2011      Invariant      7          10
//      07/10/2011      en-GB         10         7
//      07/10/2011      en-US         7          10

```

String comparison and sorting

Conventions for comparing and sorting strings vary from culture to culture. For example, the sort order may be based on phonetics or on the visual representation of characters.

In East Asian languages, characters are sorted by the stroke and radical of ideographs. Sorting also depends on the order languages and cultures use for the alphabet. For example, the Danish language has an "Æ" character that it sorts after "Z" in the alphabet. In addition, comparisons can be case-sensitive or case-insensitive, and casing rules might differ by culture. Ordinal comparison, on the other hand, uses the Unicode code points of individual characters in a string when comparing and sorting strings.

Sort rules determine the alphabetic order of Unicode characters and how two strings compare to each other. For example, the [String.Compare\(String, String, StringComparison\)](#) method compares two strings based on the [StringComparison](#) parameter. If the parameter value is [StringComparison.CurrentCulture](#), the method performs a linguistic comparison that uses the conventions of the current culture; if the parameter value is [StringComparison.OrdinalIgnoreCase](#), the method performs an ordinal comparison. Consequently, as the following example shows, if the current culture is U.S. English, the first call to the [String.Compare\(String, String, StringComparison\)](#) method (using culture-sensitive comparison) considers "a" less than "A", but the second call to the same method (using ordinal comparison) considers "a" greater than "A".

C#

```
using System;
using System.Globalization;
using System.Threading;

public class Example2
{
    public static void Main()
    {
        Thread.CurrentThread.CurrentCulture =
CultureInfo.CreateSpecificCulture("en-US");
        Console.WriteLine(String.Compare("A", "a",
StringComparison.CurrentCulture));
        Console.WriteLine(String.Compare("A", "a", StringComparison.OrdinalIgnoreCase));
    }
}
// The example displays the following output:
//      1
//     -32
```

.NET supports word, string, and ordinal sort rules:

- A word sort performs a culture-sensitive comparison of strings in which certain nonalphanumeric Unicode characters might have special weights assigned to them. For example, the hyphen (-) might have a very small weight assigned to it so that "coop" and "co-op" appear next to each other in a sorted list. For a list of the

[String methods that compare two strings using word sort rules, see the String operations by category section.](#)

- A string sort also performs a culture-sensitive comparison. It is similar to a word sort, except that there are no special cases, and all nonalphanumeric symbols come before all alphanumeric Unicode characters. Two strings can be compared using string sort rules by calling the [CompareInfo.Compare](#) method overloads that have an `options` parameter that is supplied a value of [CompareOptions.StringSort](#). Note that this is the only method that .NET provides to compare two strings using string sort rules.
- An ordinal sort compares strings based on the numeric value of each [Char](#) object in the string. An ordinal comparison is automatically case-sensitive because the lowercase and uppercase versions of a character have different code points. However, if case is not important, you can specify an ordinal comparison that ignores case. This is equivalent to converting the string to uppercase by using the invariant culture and then performing an ordinal comparison on the result. For a list of the [String](#) methods that compare two strings using ordinal sort rules, see the [String operations by category section](#).

A culture-sensitive comparison is any comparison that explicitly or implicitly uses a [CultureInfo](#) object, including the invariant culture that is specified by the [CultureInfo.InvariantCulture](#) property. The implicit culture is the current culture, which is specified by the [Thread.CurrentCulture](#) and [CultureInfo.CurrentCulture](#) properties. There is considerable variation in the sort order of alphabetic characters (that is, characters for which the [Char.IsLetter](#) property returns `true`) across cultures. You can specify a culture-sensitive comparison that uses the conventions of a specific culture by supplying a [CultureInfo](#) object to a string comparison method such as [Compare\(String, String, CultureInfo, CompareOptions\)](#). You can specify a culture-sensitive comparison that uses the conventions of the current culture by supplying [StringComparison.CurrentCulture](#), [StringComparison.CurrentCultureIgnoreCase](#), or any member of the [CompareOptions](#) enumeration other than [CompareOptions.Ordinal](#) or [CompareOptions.OrdinalIgnoreCase](#) to an appropriate overload of the [Compare](#) method. A culture-sensitive comparison is generally appropriate for sorting whereas an ordinal comparison is not. An ordinal comparison is generally appropriate for determining whether two strings are equal (that is, for determining identity) whereas a culture-sensitive comparison is not.

The following example illustrates the difference between culture-sensitive and ordinal comparison. The example evaluates three strings, "Apple", "Æble", and "AEble", using ordinal comparison and the conventions of the da-DK and en-US cultures (each of which is the default culture at the time the [Compare](#) method is called). Because the

Danish language treats the character "Æ" as an individual letter and sorts it after "Z" in the alphabet, the string "Æble" is greater than "Apple". However, "Æble" is not considered equivalent to "AEble", so "Æble" is also greater than "AEble". The en-US culture doesn't include the letter "Æ" but treats it as equivalent to "AE", which explains why "Æble" is less than "Apple" but equal to "AEble". Ordinal comparison, on the other hand, considers "Apple" to be less than "Æble", and "Æble" to be greater than "AEble".

C#

```
using System;
using System.Globalization;
using System.Threading;

public class CompareStringSample
{
    public static void Main()
    {
        string str1 = "Apple";
        string str2 = "Æble";
        string str3 = "AEble";

        // Set the current culture to Danish in Denmark.
        Thread.CurrentCulture = new CultureInfo("da-DK");
        Console.WriteLine("Current culture: {0}",
                          CultureInfo.CurrentCulture.Name);
        Console.WriteLine("Comparison of {0} with {1}: {2}",
                          str1, str2, String.Compare(str1, str2));
        Console.WriteLine("Comparison of {0} with {1}: {2}\n",
                          str2, str3, String.Compare(str2, str3));

        // Set the current culture to English in the U.S.
        Thread.CurrentCulture = new CultureInfo("en-US");
        Console.WriteLine("Current culture: {0}",
                          CultureInfo.CurrentCulture.Name);
        Console.WriteLine("Comparison of {0} with {1}: {2}",
                          str1, str2, String.Compare(str1, str2));
        Console.WriteLine("Comparison of {0} with {1}: {2}\n",
                          str2, str3, String.Compare(str2, str3));

        // Perform an ordinal comparison.
        Console.WriteLine("Ordinal comparison");
        Console.WriteLine("Comparison of {0} with {1}: {2}",
                          str1, str2,
                          String.Compare(str1, str2,
                                         StringComparison.Ordinal));
        Console.WriteLine("Comparison of {0} with {1}: {2}",
                          str2, str3,
                          String.Compare(str2, str3,
                                         StringComparison.Ordinal));
    }
}

// The example displays the following output:
```

```
//      Current culture: da-DK
//      Comparison of Apple with Äble: -1
//      Comparison of Äble with AEble: 1
//
//      Current culture: en-US
//      Comparison of Apple with Äble: 1
//      Comparison of Äble with AEble: 0
//
//      Ordinal comparison
//      Comparison of Apple with Äble: -133
//      Comparison of Äble with AEble: 133
```

Use the following general guidelines to choose an appropriate sorting or string comparison method:

- If you want the strings to be ordered based on the user's culture, you should order them based on the conventions of the current culture. If the user's culture changes, the order of sorted strings will also change accordingly. For example, a thesaurus application should always sort words based on the user's culture.
- If you want the strings to be ordered based on the conventions of a specific culture, you should order them by supplying a [CultureInfo](#) object that represents that culture to a comparison method. For example, in an application designed to teach students a particular language, you want strings to be ordered based on the conventions of one of the cultures that speaks that language.
- If you want the order of strings to remain unchanged across cultures, you should order them based on the conventions of the invariant culture or use an ordinal comparison. For example, you would use an ordinal sort to organize the names of files, processes, mutexes, or named pipes.
- For a comparison that involves a security decision (such as whether a username is valid), you should always perform an ordinal test for equality by calling an overload of the [Equals](#) method.

Note

The culture-sensitive sorting and casing rules used in string comparison depend on the version of the .NET. On .NET Core, string comparison depends on the version of the Unicode Standard supported by the underlying operating system. In .NET Framework 4.5 and later versions running on Windows 8 or later, sorting, casing, normalization, and Unicode character information conform to the Unicode 6.0 standard. On other Windows operating systems, they conform to the Unicode 5.0 standard.

For more information about word, string, and ordinal sort rules, see the [System.Globalization.CompareOptions](#) topic. For additional recommendations on when to use each rule, see [Best Practices for Using Strings](#).

Ordinarily, you don't call string comparison methods such as [Compare](#) directly to determine the sort order of strings. Instead, comparison methods are called by sorting methods such as [Array.Sort](#) or [List<T>.Sort](#). The following example performs four different sorting operations (word sort using the current culture, word sort using the invariant culture, ordinal sort, and string sort using the invariant culture) without explicitly calling a string comparison method, although they do specify the type of comparison to use. Note that each type of sort produces a unique ordering of strings in its array.

C#

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Globalization;

public class Example3
{
    public static void Main()
    {
        string[] strings = { "coop", "co-op", "cooperative",
                             "co\u00e2operative", "cœur", "coeur" };

        // Perform a word sort using the current (en-US) culture.
        string[] current = new string[strings.Length];
        strings.CopyTo(current, 0);
        Array.Sort(current, StringComparer.CurrentCulture);

        // Perform a word sort using the invariant culture.
        string[] invariant = new string[strings.Length];
        strings.CopyTo(invariant, 0);
        Array.Sort(invariant, StringComparer.InvariantCulture);

        // Perform an ordinal sort.
        string[] ordinal = new string[strings.Length];
        strings.CopyTo(ordinal, 0);
        Array.Sort(ordinal, StringComparer.Ordinal);

        // Perform a string sort using the current culture.
        string[] stringSort = new string[strings.Length];
        strings.CopyTo(stringSort, 0);
        Array.Sort(stringSort, new SCompare());

        // Display array values
        Console.WriteLine("{0,13} {1,13} {2,15} {3,13} {4,13}\n",
                         "Original", "Word Sort", "Invariant Word",
                         "Ordinal Sort", "String Sort");
    }
}
```

```

        for (int ctr = 0; ctr < strings.Length; ctr++)
            Console.WriteLine("{0,13} {1,13} {2,15} {3,13} {4,13}",
                strings[ctr], current[ctr], invariant[ctr],
                ordinal[ctr], stringSort[ctr] );
    }
}

// IComparer<String> implementation to perform string sort.
internal class SCompare : IComparer<String>
{
    public int Compare(string x, string y)
    {
        return CultureInfo.CurrentCulture.CompareInfo.Compare(x, y,
CompareOptions.StringSort);
    }
}
// The example displays the following output:
//          Original      Word Sort  Invariant Word  Ordinal Sort  String
Sort
//
//          coop           cœur          cœur          co-op         co-
op
//
//          co-op          cœur          cœur          cœur
cœur
//
//          cooperative     coop          coop          coop
coeur
//
//          cooperative     co-op          co-op        cooperative       coop
//
//          cœur          cooperative   cooperative  cooperative  cooperative
//
//          cœur          cooperative   cooperative  cœur        cooperative

```

Tip

Internally, .NET uses sort keys to support culturally sensitive string comparison. Each character in a string is given several categories of sort weights, including alphabetic, case, and diacritic. A sort key, represented by the **SortKey** class, provides a repository of these weights for a particular string. If your app performs a large number of searching or sorting operations on the same set of strings, you can improve its performance by generating and storing sort keys for all the strings that it uses. When a sort or comparison operation is required, you use the sort keys instead of the strings. For more information, see the **SortKey** class.

If you don't specify a string comparison convention, sorting methods such as [Array.Sort\(Array\)](#) perform a culture-sensitive, case-sensitive sort on strings. The following example illustrates how changing the current culture affects the order of sorted strings in an array. It creates an array of three strings. First, it sets the [System.Threading.Thread.CurrentCulture](#) property to en-US and calls the [Array.Sort\(Array\)](#) method. The resulting sort order is based on sorting conventions for

the English (United States) culture. Next, the example sets the `System.Threading.Thread.CurrentCulture` property to da-DK and calls the `Array.Sort` method again. Notice how the resulting sort order differs from the en-US results because it uses the sorting conventions for Danish (Denmark).

C#

```
using System;
using System.Globalization;
using System.Threading;

public class ArraySort
{
    public static void Main(String[] args)
    {
        // Create and initialize a new array to store the strings.
        string[] stringArray = { "Apple", "Æble", "Zebra" };

        // Display the values of the array.
        Console.WriteLine( "The original string array:");
        PrintIndexAndValues(stringArray);

        // Set the CurrentCulture to "en-US".
        Thread.CurrentCulture = new CultureInfo("en-US");
        // Sort the values of the array.
        Array.Sort(stringArray);

        // Display the values of the array.
        Console.WriteLine("After sorting for the culture \"en-US\":");
        PrintIndexAndValues(stringArray);

        // Set the CurrentCulture to "da-DK".
        Thread.CurrentCulture = new CultureInfo("da-DK");
        // Sort the values of the Array.
        Array.Sort(stringArray);

        // Display the values of the array.
        Console.WriteLine("After sorting for the culture \"da-DK\":");
        PrintIndexAndValues(stringArray);
    }

    public static void PrintIndexAndValues(string[] myArray)
    {
        for (int i = myArray.GetLowerBound(0); i <=
             myArray.GetUpperBound(0); i++ )
            Console.WriteLine("[{0}]: {1}", i, myArray[i]);
        Console.WriteLine();
    }
}

// The example displays the following output:
//      The original string array:
//      [0]: Apple
//      [1]: Æble
//      [2]: Zebra
```

```
//      After sorting for the "en-US" culture:  
//      [0]: Åble  
//      [1]: Apple  
//      [2]: Zebra  
  
//      After sorting for the culture "da-DK":  
//      [0]: Apple  
//      [1]: Zebra  
//      [2]: Åble
```

⚠ Warning

If your primary purpose in comparing strings is to determine whether they are equal, you should call the [String.Equals](#) method. Typically, you should use [Equals](#) to perform an ordinal comparison. The [String.Compare](#) method is intended primarily to sort strings.

String search methods, such as [String.StartsWith](#) and [String.IndexOf](#), also can perform culture-sensitive or ordinal string comparisons. The following example illustrates the differences between ordinal and culture-sensitive comparisons using the [IndexOf](#) method. A culture-sensitive search in which the current culture is English (United States) considers the substring "oe" to match the ligature "œ". Because a soft hyphen (U+00AD) is a zero-width character, the search treats the soft hyphen as equivalent to [String.Empty](#) and finds a match at the beginning of the string. An ordinal search, on the other hand, does not find a match in either case.

C#

```
using System;  
  
public class Example8  
{  
    public static void Main()  
    {  
        // Search for "oe" and "œu" in "œufs" and "oeufs".  
        string s1 = "œufs";  
        string s2 = "oeufs";  
        FindInString(s1, "oe", StringComparison.CurrentCulture);  
        FindInString(s1, "oe", StringComparison.Ordinal);  
        FindInString(s2, "œu", StringComparison.CurrentCulture);  
        FindInString(s2, "œu", StringComparison.Ordinal);  
        Console.WriteLine();  
  
        string s3 = "co\u00ADoperative";  
        FindInString(s3, "\u00AD", StringComparison.CurrentCulture);  
        FindInString(s3, "\u00AD", StringComparison.Ordinal);  
    }  
}
```

```

private static void FindInString(string s, string substring,
 StringComparison options)
{
    int result = s.IndexOf(substring, options);
    if (result != -1)
        Console.WriteLine("'{}' found in {} at position {}",
                         substring, s, result);
    else
        Console.WriteLine("'{}' not found in {}", substring, s);
}
// The example displays the following output:
//      'oe' found in œufs at position 0
//      'oe' not found in œufs
//      'œu' found in œufs at position 0
//      'œu' not found in œufs
//
//      '' found in cooperative at position 0
//      '' found in cooperative at position 2

```

Search in strings

String search methods, such as [String.StartsWith](#) and [String.IndexOf](#), also can perform culture-sensitive or ordinal string comparisons to determine whether a character or substring is found in a specified string.

The search methods in the [String](#) class that search for an individual character, such as the [IndexOf](#) method, or one of a set of characters, such as the [IndexOfAny](#) method, all perform an ordinal search. To perform a culture-sensitive search for a character, you must call a [CompareInfo](#) method such as [CompareInfo.IndexOf\(String, Char\)](#) or [CompareInfo.LastIndexOf\(String, Char\)](#). Note that the results of searching for a character using ordinal and culture-sensitive comparison can be very different. For example, a search for a precomposed Unicode character such as the ligature "Æ" (U+00C6) might match any occurrence of its components in the correct sequence, such as "AE" (U+041U+0045), depending on the culture. The following example illustrates the difference between the [String.IndexOf\(Char\)](#) and [CompareInfo.IndexOf\(String, Char\)](#) methods when searching for an individual character. The ligature "æ" (U+00E6) is found in the string "aerial" when using the conventions of the en-US culture, but not when using the conventions of the da-DK culture or when performing an ordinal comparison.

C#

```

using System;
using System.Globalization;

```

```

public class Example17
{
    public static void Main()
    {
        String[] cultureNames = { "da-DK", "en-US" };
        CompareInfo ci;
        String str = "aerial";
        Char ch = 'æ'; // U+00E6

        Console.Write("Ordinal comparison -- ");
        Console.WriteLine("Position of '{0}' in {1}: {2}", ch, str,
                         str.IndexOf(ch));

        foreach (var cultureName in cultureNames) {
            ci = CultureInfo.CreateSpecificCulture(cultureName).CompareInfo;
            Console.Write("{0} cultural comparison -- ", cultureName);
            Console.WriteLine("Position of '{0}' in {1}: {2}", ch, str,
                             ci.IndexOf(str, ch));
        }
    }
}

// The example displays the following output:
//      Ordinal comparison -- Position of 'æ' in aerial: -1
//      da-DK cultural comparison -- Position of 'æ' in aerial: -1
//      en-US cultural comparison -- Position of 'æ' in aerial: 0

```

On the other hand, [String](#) class methods that search for a string rather than a character perform a culture-sensitive search if search options are not explicitly specified by a parameter of type [StringComparison](#). The sole exception is [Contains](#), which performs an ordinal search.

Test for equality

Use the [String.Compare](#) method to determine the relationship of two strings in the sort order. Typically, this is a culture-sensitive operation. In contrast, call the [String.Equals](#) method to test for equality. Because the test for equality usually compares user input with some known string, such as a valid user name, a password, or a file system path, it is typically an ordinal operation.

Warning

It is possible to test for equality by calling the [String.Compare](#) method and determining whether the return value is zero. However, this practice is not recommended. To determine whether two strings are equal, you should call one of the overloads of the [String.Equals](#) method. The preferred overload to call is either the instance [Equals\(String, StringComparison\)](#) method or the static [Equals\(String, String, StringComparison\)](#) method, because both methods include a

System.StringComparison parameter that explicitly specifies the type of comparison.

The following example illustrates the danger of performing a culture-sensitive comparison for equality when an ordinal one should be used instead. In this case, the intent of the code is to prohibit file system access from URLs that begin with "FILE://" or "file://" by performing a case-insensitive comparison of the beginning of a URL with the string "FILE://". However, if a culture-sensitive comparison is performed using the Turkish (Turkey) culture on a URL that begins with "file://", the comparison for equality fails, because the Turkish uppercase equivalent of the lowercase "i" is "İ" instead of "I". As a result, file system access is inadvertently permitted. On the other hand, if an ordinal comparison is performed, the comparison for equality succeeds, and file system access is denied.

C#

```
using System;
using System.Globalization;
using System.Threading;

public class Example4
{
    public static void Main()
    {
        Thread.CurrentThread.CurrentCulture =
CultureInfo.CreateSpecificCulture("tr-TR");

        string filePath = "file:///c:/notes.txt";

        Console.WriteLine("Culture-sensitive test for equality:");
        if (! TestForEquality(filePath,
StringComparison.CurrentCultureIgnoreCase))
            Console.WriteLine("Access to {0} is allowed.", filePath);
        else
            Console.WriteLine("Access to {0} is not allowed.", filePath);

        Console.WriteLine("\nOrdinal test for equality:");
        if (! TestForEquality(filePath, StringComparison.OrdinalIgnoreCase))
            Console.WriteLine("Access to {0} is allowed.", filePath);
        else
            Console.WriteLine("Access to {0} is not allowed.", filePath);
    }

    private static bool TestForEquality(string str, StringComparison cmp)
    {
        int position = str.IndexOf(":/");
        if (position < 0) return false;

        string substring = str.Substring(0, position);
```

```
        return substring.Equals("FILE", cmp);
    }
}
// The example displays the following output:
//     Culture-sensitive test for equality:
//         Access to file:///c:/notes.txt is allowed.
//
//     Ordinal test for equality:
//         Access to file:///c:/notes.txt is not allowed.
```

Normalization

Some Unicode characters have multiple representations. For example, any of the following code points can represent the letter "å":

- U+1EAF
- U+0103 U+0301
- U+0061 U+0306 U+0301

Multiple representations for a single character complicate searching, sorting, matching, and other string operations.

The Unicode standard defines a process called normalization that returns one binary representation of a Unicode character for any of its equivalent binary representations. Normalization can use several algorithms, called normalization forms, that follow different rules. .NET supports Unicode normalization forms C, D, KC, and KD. When strings have been normalized to the same normalization form, they can be compared by using ordinal comparison.

An ordinal comparison is a binary comparison of the Unicode scalar value of corresponding [Char](#) objects in each string. The [String](#) class includes a number of methods that can perform an ordinal comparison, including the following:

- Any overload of the [Compare](#), [Equals](#), [StartsWith](#), [EndsWith](#), [IndexOf](#), and [LastIndexOf](#) methods that includes a [StringComparison](#) parameter. The method performs an ordinal comparison if you supply a value of [StringComparison.Ordinal](#) or [StringComparison.OrdinalIgnoreCase](#) for this parameter.
- The overloads of the [CompareOrdinal](#) method.
- Methods that use ordinal comparison by default, such as [Contains](#), [Replace](#), and [Split](#).
- Methods that search for a [Char](#) value or for the elements in a [Char](#) array in a string instance. Such methods include [IndexOf\(Char\)](#) and [Split\(Char\[\]\)](#).

You can determine whether a string is normalized to normalization form C by calling the [String.IsNormalized\(\)](#) method, or you can call the [String.IsNormalized\(NormalizationForm\)](#) method to determine whether a string is normalized to a specified normalization form. You can also call the [String.Normalize\(\)](#) method to convert a string to normalization form C, or you can call the [String.Normalize\(NormalizationForm\)](#) method to convert a string to a specified normalization form. For step-by-step information about normalizing and comparing strings, see the [Normalize\(\)](#) and [Normalize\(NormalizationForm\)](#) methods.

The following simple example illustrates string normalization. It defines the letter "ő" in three different ways in three different strings, and uses an ordinal comparison for equality to determine that each string differs from the other two strings. It then converts each string to the supported normalization forms, and again performs an ordinal comparison of each string in a specified normalization form. In each case, the second test for equality shows that the strings are equal.

C#

```
using System;
using System.Globalization;
using System.IO;
using System.Text;

public class Example13
{
    private static StreamWriter sw;

    public static void Main()
    {
        sw = new StreamWriter(@".\TestNorm1.txt");

        // Define three versions of the same word.
        string s1 = "sőng";           // create word with U+1ED1
        string s2 = "s\u00F4ng";
        string s3 = "s\u0302ng";

        TestForEquality(s1, s2, s3);
        sw.WriteLine();

        // Normalize and compare strings using each normalization form.
        foreach (string formName in Enum.GetNames(typeof(NormalizationForm)))
        {
            sw.WriteLine("Normalization {0}:\n", formName);
            NormalizationForm nf = (NormalizationForm)
                Enum.Parse(typeof(NormalizationForm), formName);
            string[] sn = NormalizeStrings(nf, s1, s2, s3);
            TestForEquality(sn);
            sw.WriteLine("\n");
        }
    }

    private void TestForEquality(string[] strings)
    {
        for (int i = 0; i < strings.Length - 1; i++)
        {
            if (strings[i] != strings[i + 1])
            {
                sw.WriteLine("The strings are not equal.");
                return;
            }
        }
        sw.WriteLine("The strings are equal.");
    }

    private string[] NormalizeStrings(NormalizationForm nf, string s1, string s2, string s3)
    {
        string[] result = new string[3];
        result[0] = String.Normalize(nf, s1);
        result[1] = String.Normalize(nf, s2);
        result[2] = String.Normalize(nf, s3);
        return result;
    }
}
```

```

        sw.Close();
    }

private static void TestForEquality(params string[] words)
{
    for (int ctr = 0; ctr <= words.Length - 2; ctr++)
        for (int ctr2 = ctr + 1; ctr2 <= words.Length - 1; ctr2++)
            sw.WriteLine("{0} ({1}) = {2} ({3}): {4}",
                        words[ctr], ShowBytes(words[ctr]),
                        words[ctr2], ShowBytes(words[ctr2]),
                        words[ctr].Equals(words[ctr2],
StringComparison.OrdinalIgnoreCase));
}

private static string ShowBytes(string str)
{
    string result = null;
    foreach (var ch in str)
        result += $"{(ushort)ch:X4} ";
    return result.Trim();
}

private static string[] NormalizeStrings(NormalizationForm nf, params
string[] words)
{
    for (int ctr = 0; ctr < words.Length; ctr++)
        if (!words[ctr].IsNormalized(nf))
            words[ctr] = words[ctr].Normalize(nf);
    return words;
}

// The example displays the following output:
//      sông (0073 1ED1 006E 0067) = sông (0073 00F4 0301 006E 0067): False
//      sông (0073 1ED1 006E 0067) = sông (0073 006F 0302 0301 006E 0067):
False
//      sông (0073 00F4 0301 006E 0067) = sông (0073 006F 0302 0301 006E
0067): False
//
//      Normalization FormC:
//
//      sông (0073 1ED1 006E 0067) = sông (0073 1ED1 006E 0067): True
//      sông (0073 1ED1 006E 0067) = sông (0073 1ED1 006E 0067): True
//      sông (0073 1ED1 006E 0067) = sông (0073 1ED1 006E 0067): True
//
//
//      Normalization FormD:
//
//      sông (0073 006F 0302 0301 006E 0067) = sông (0073 006F 0302 0301
006E 0067): True
//      sông (0073 006F 0302 0301 006E 0067) = sông (0073 006F 0302 0301
006E 0067): True
//      sông (0073 006F 0302 0301 006E 0067) = sông (0073 006F 0302 0301
006E 0067): True
//
//

```

```
//      Normalization FormKC:  
//  
//      sông (0073 1ED1 006E 0067) = sông (0073 1ED1 006E 0067): True  
//      sông (0073 1ED1 006E 0067) = sông (0073 1ED1 006E 0067): True  
//      sông (0073 1ED1 006E 0067) = sông (0073 1ED1 006E 0067): True  
//  
//  
//      Normalization FormKD:  
//  
//      sông (0073 006F 0302 0301 006E 0067) = sông (0073 006F 0302 0301  
006E 0067): True  
//      sông (0073 006F 0302 0301 006E 0067) = sông (0073 006F 0302 0301  
006E 0067): True  
//      sông (0073 006F 0302 0301 006E 0067) = sông (0073 006F 0302 0301  
006E 0067): True
```

For more information about normalization and normalization forms, see [System.Text.NormalizationForm](#), as well as [Unicode Standard Annex #15: Unicode Normalization Forms](#) and the [Normalization FAQ](#) on the unicode.org website.

String operations by category

The [String](#) class provides members for comparing strings, testing strings for equality, finding characters or substrings in a string, modifying a string, extracting substrings from a string, combining strings, formatting values, copying a string, and normalizing a string.

Compare strings

You can compare strings to determine their relative position in the sort order by using the following [String](#) methods:

- [Compare](#) returns an integer that indicates the relationship of one string to a second string in the sort order.
- [CompareOrdinal](#) returns an integer that indicates the relationship of one string to a second string based on a comparison of their code points.
- [CompareTo](#) returns an integer that indicates the relationship of the current string instance to a second string in the sort order. The [CompareTo\(String\)](#) method provides the [IComparable](#) and [IComparable<T>](#) implementations for the [String](#) class.

Test strings for equality

You call the [Equals](#) method to determine whether two strings are equal. The instance [Equals\(String, String, StringComparison\)](#) and the static [Equals\(String, StringComparison\)](#) overloads let you specify whether the comparison is culture-sensitive or ordinal, and whether case is considered or ignored. Most tests for equality are ordinal, and comparisons for equality that determine access to a system resource (such as a file system object) should always be ordinal.

Find characters in a string

The [String](#) class includes two kinds of search methods:

- Methods that return a [Boolean](#) value to indicate whether a particular substring is present in a string instance. These include the [Contains](#), [EndsWith](#), and [StartsWith](#) methods.
- Methods that indicate the starting position of a substring in a string instance. These include the [IndexOf](#), [IndexOfAny](#), [LastIndexOf](#), and [LastIndexOfAny](#) methods.

Warning

If you want to search a string for a particular pattern rather than a specific substring, you should use regular expressions. For more information, see [.NET Regular Expressions](#).

Modify a string

The [String](#) class includes the following methods that appear to modify the value of a string:

- [Insert](#) inserts a string into the current [String](#) instance.
- [PadLeft](#) inserts one or more occurrences of a specified character at the beginning of a string.
- [PadRight](#) inserts one or more occurrences of a specified character at the end of a string.
- [Remove](#) deletes a substring from the current [String](#) instance.
- [Replace](#) replaces a substring with another substring in the current [String](#) instance.
- [ToLower](#) and [ToLowerInvariant](#) convert all the characters in a string to lowercase.

- [ToUpper](#) and [ToUpperInvariant](#) convert all the characters in a string to uppercase.
- [Trim](#) removes all occurrences of a character from the beginning and end of a string.
- [TrimEnd](#) removes all occurrences of a character from the end of a string.
- [TrimStart](#) removes all occurrences of a character from the beginning of a string.

Important

All string modification methods return a new [String](#) object. They don't modify the value of the current instance.

Extract substrings from a string

The [String.Split](#) method separates a single string into multiple strings. Overloads of the method allow you to specify multiple delimiters, to limit the number of substrings that the method extracts, to trim white space from substrings, and to specify whether empty strings (which occur when delimiters are adjacent) are included among the returned strings.

Combine strings

The following [String](#) methods can be used for string concatenation:

- [Concat](#) combines one or more substrings into a single string.
- [Join](#) concatenates one or more substrings into a single element and adds a separator between each substring.

Format values

The [String.Format](#) method uses the composite formatting feature to replace one or more placeholders in a string with the string representation of some object or value. The [Format](#) method is often used to do the following:

- To embed the string representation of a numeric value in a string.
- To embed the string representation of a date and time value in a string.
- To embed the string representation of an enumeration value in a string.
- To embed the string representation of some object that supports the [IFormattable](#) interface in a string.
- To right-justify or left-justify a substring in a field within a larger string.

For detailed information about formatting operations and examples, see the [Format](#) overload summary.

Copy a string

You can call the following [String](#) methods to make a copy of a string:

- [Clone](#) returns a reference to an existing [String](#) object.
- [Copy](#) creates a copy of an existing string.
- [CopyTo](#) copies a portion of a string to a character array.

Normalize a string

In Unicode, a single character can have multiple code points. Normalization converts these equivalent characters into the same binary representation. The [String.Normalize](#) method performs the normalization, and the [String.IsNormalized](#) method determines whether a string is normalized.

For more information and an example, see the [Normalization](#) section earlier in this article.

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



.NET feedback

.NET is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

System.Char struct

Article • 01/08/2024

This article provides supplementary remarks to the reference documentation for this API.

The [Char](#) structure represents Unicode code points by using UTF-16 encoding. The value of a [Char](#) object is its 16-bit numeric (ordinal) value.

If you aren't familiar with Unicode, scalar values, code points, surrogate pairs, UTF-16, and the [Rune](#) type, see [Introduction to character encoding in .NET](#).

This article examines the relationship between a [Char](#) object and a character and discuss some common tasks performed with [Char](#) instances. We recommend that you consider the [Rune](#) type, introduced in .NET Core 3.0, as an alternative to [Char](#) for performing some of these tasks.

Char objects, Unicode characters, and strings

A [String](#) object is a sequential collection of [Char](#) structures that represents a string of text. Most Unicode characters can be represented by a single [Char](#) object, but a character that is encoded as a base character, surrogate pair, and/or combining character sequence is represented by multiple [Char](#) objects. For this reason, a [Char](#) structure in a [String](#) object is not necessarily equivalent to a single Unicode character.

Multiple 16-bit code units are used to represent single Unicode characters in the following cases:

- Glyphs, which may consist of a single character or of a base character followed by one or more combining characters. For example, the character ä is represented by a [Char](#) object whose code unit is U+0061 followed by a [Char](#) object whose code unit is U+0308. (The character ä can also be defined by a single [Char](#) object that has a code unit of U+00E4.) The following example illustrates that the character ä consists of two [Char](#) objects.

C#

```
using System;
using System.IO;

public class Example1
{
    public static void Main()
```

```

    }
    StreamWriter sw = new StreamWriter("chars1.txt");
    char[] chars = { '\u0061', '\u0308' };
    string strng = new String(chars);
    sw.WriteLine(strng);
    sw.Close();
}
}

// The example produces the following output:
//      ä

```

- Characters outside the Unicode Basic Multilingual Plane (BMP). Unicode supports sixteen planes in addition to the BMP, which represents plane 0. A Unicode code point is represented in UTF-32 by a 21-bit value that includes the plane. For example, U+1D160 represents the MUSICAL SYMBOL EIGHTH NOTE character. Because UTF-16 encoding has only 16 bits, characters outside the BMP are represented by surrogate pairs in UTF-16. The following example illustrates that the UTF-32 equivalent of U+1D160, the MUSICAL SYMBOL EIGHTH NOTE character, is U+D834 U+DD60. U+D834 is the high surrogate; high surrogates range from U+D800 through U+DBFF. U+DD60 is the low surrogate; low surrogates range from U+DC00 through U+DFFF.

C#

```

using System;
using System.IO;

public class Example3
{
    public static void Main()
    {
        StreamWriter sw = new StreamWriter(@".\chars2.txt");
        int utf32 = 0x1D160;
        string surrogate = Char.ConvertFromUtf32(utf32);
        sw.WriteLine("U+{0:X6} UTF-32 = {1} ({2}) UTF-16",
                    utf32, surrogate, ShowCodePoints(surrogate));
        sw.Close();
    }

    private static string ShowCodePoints(string value)
    {
        string retval = null;
        foreach (var ch in value)
            retval += String.Format("U+{0:X4} ", Convert.ToInt16(ch));

        return retval.Trim();
    }
}

```

```
// The example produces the following output:  
//      U+01D160 UTF-32 = δ (U+D834 U+DD60) UTF-16
```

Characters and character categories

Each Unicode character or valid surrogate pair belongs to a Unicode category. In .NET, Unicode categories are represented by members of the [UnicodeCategory](#) enumeration and include values such as [UnicodeCategory.CurrencySymbol](#), [UnicodeCategory.LowercaseLetter](#), and [UnicodeCategory.SpaceSeparator](#), for example.

To determine the Unicode category of a character, call the [GetUnicodeCategory](#) method. For example, the following example calls the [GetUnicodeCategory](#) to display the Unicode category of each character in a string. The example works correctly only if there are no surrogate pairs in the [String](#) instance.

C#

```
using System;  
using System.Globalization;  
  
class Example  
{  
    public static void Main()  
    {  
        // Define a string with a variety of character categories.  
        String s = "The red car drove down the long, narrow, secluded road.";  
        // Determine the category of each character.  
        foreach (var ch in s)  
            Console.WriteLine("{0}': {1}", ch, Char.GetUnicodeCategory(ch));  
    }  
}  
// The example displays the following output:  
//      'T': UppercaseLetter  
//      'h': LowercaseLetter  
//      'e': LowercaseLetter  
//      ' ': SpaceSeparator  
//      'r': LowercaseLetter  
//      'e': LowercaseLetter  
//      'd': LowercaseLetter  
//      ' ': SpaceSeparator  
//      'c': LowercaseLetter  
//      'a': LowercaseLetter  
//      'r': LowercaseLetter  
//      ' ': SpaceSeparator  
//      'd': LowercaseLetter  
//      'r': LowercaseLetter  
//      'o': LowercaseLetter  
//      'v': LowercaseLetter  
//      'e': LowercaseLetter
```

```
//      ' ': SpaceSeparator
//      'd': LowercaseLetter
//      'o': LowercaseLetter
//      'w': LowercaseLetter
//      'n': LowercaseLetter
//      ' ': SpaceSeparator
//      't': LowercaseLetter
//      'h': LowercaseLetter
//      'e': LowercaseLetter
//      ' ': SpaceSeparator
//      'l': LowercaseLetter
//      'o': LowercaseLetter
//      'n': LowercaseLetter
//      'g': LowercaseLetter
//      ',': OtherPunctuation
//      ' ': SpaceSeparator
//      'n': LowercaseLetter
//      'a': LowercaseLetter
//      'r': LowercaseLetter
//      'r': LowercaseLetter
//      'o': LowercaseLetter
//      'w': LowercaseLetter
//      ',': OtherPunctuation
//      ' ': SpaceSeparator
//      's': LowercaseLetter
//      'e': LowercaseLetter
//      'c': LowercaseLetter
//      'l': LowercaseLetter
//      'u': LowercaseLetter
//      'd': LowercaseLetter
//      'e': LowercaseLetter
//      'd': LowercaseLetter
//      ' ': SpaceSeparator
//      'r': LowercaseLetter
//      'o': LowercaseLetter
//      'a': LowercaseLetter
//      'd': LowercaseLetter
//      '.': OtherPunctuation
```

Internally, for characters outside the ASCII range (U+0000 through U+00FF), the [GetUnicodeCategory](#) method depends on Unicode categories reported by the [CharUnicodeInfo](#) class. Starting with .NET Framework 4.6.2, Unicode characters are classified based on [The Unicode Standard, Version 8.0.0](#). In versions of the .NET Framework from .NET Framework 4 to .NET Framework 4.6.1, they are classified based on [The Unicode Standard, Version 6.3.0](#).

Characters and text elements

Because a single character can be represented by multiple [Char](#) objects, it is not always meaningful to work with individual [Char](#) objects. For instance, the following example

converts the Unicode code points that represent the Aegean numbers zero through 9 to UTF-16 encoded code units. Because it erroneously equates [Char](#) objects with characters, it inaccurately reports that the resulting string has 20 characters.

C#

```
using System;

public class Example5
{
    public static void Main()
    {
        string result = String.Empty;
        for (int ctr = 0x10107; ctr <= 0x10110; ctr++) // Range of Aegean
numbers.
            result += Char.ConvertFromUtf32(ctr);

        Console.WriteLine("The string contains {0} characters.", result.Length);
    }
}
// The example displays the following output:
//      The string contains 20 characters.
```

You can do the following to avoid the assumption that a [Char](#) object represents a single character:

- You can work with a [String](#) object in its entirety instead of working with its individual characters to represent and analyze linguistic content.
- You can use [String.EnumerateRunes](#) as shown in the following example:

C#

```
int CountLetters(string s)
{
    int letterCount = 0;

    foreach (Rune rune in s.EnumerateRunes())
    {
        if (Rune.IsLetter(rune))
            { letterCount++; }
    }

    return letterCount;
}
```

- You can use the [StringInfo](#) class to work with text elements instead of individual [Char](#) objects. The following example uses the [StringInfo](#) object to count the

number of text elements in a string that consists of the Aegean numbers zero through nine. Because it considers a surrogate pair a single character, it correctly reports that the string contains ten characters.

```
C#  
  
using System;  
using System.Globalization;  
  
public class Example4  
{  
    public static void Main()  
    {  
        string result = String.Empty;  
        for (int ctr = 0x10107; ctr <= 0x10110; ctr++) // Range of Aegean numbers.  
            result += Char.ConvertFromUtf32(ctr);  
  
        StringInfo si = new StringInfo(result);  
        Console.WriteLine("The string contains {0} characters.",  
                          si.LengthInTextElements);  
    }  
}  
// The example displays the following output:  
//     The string contains 10 characters.
```

- If a string contains a base character that has one or more combining characters, you can call the `String.Normalize` method to convert the substring to a single UTF-16 encoded code unit. The following example calls the `String.Normalize` method to convert the base character U+0061 (LATIN SMALL LETTER A) and combining character U+0308 (COMBINING DIAERESIS) to U+00E4 (LATIN SMALL LETTER A WITH DIAERESIS).

```
C#  
  
using System;  
  
public class Example2  
{  
    public static void Main()  
    {  
        string combining = "\u0061\u0308";  
        ShowString(combining);  
  
        string normalized = combining.Normalize();  
        ShowString(normalized);  
    }  
  
    private static void ShowString(string s)  
    {
```

```

        Console.WriteLine("Length of string: {0} ({", s.Length);
        for (int ctr = 0; ctr < s.Length; ctr++)
        {
            Console.Write("U+{0:X4}", Convert.ToInt16(s[ctr]));
            if (ctr != s.Length - 1) Console.Write(" ");
        }
        Console.WriteLine(")\n");
    }
}

// The example displays the following output:
//      Length of string: 2 (U+0061 U+0308)
//
//      Length of string: 1 (U+00E4)

```

Common operations

The [Char](#) structure provides methods to compare [Char](#) objects, convert the value of the current [Char](#) object to an object of another type, and determine the Unicode category of a [Char](#) object:

[Expand table](#)

| To do this | Use these <code>System.Char</code> methods |
|--|--|
| Compare Char objects | CompareTo and Equals |
| Convert a code point to a string | ConvertFromUtf32 See also the Rune type. |
| Convert a Char object or a surrogate pair of Char objects to a code point | For a single character: Convert.ToInt32(Char) For a surrogate pair or a character in a string: Char.ConvertToUtf32 See also the Rune type. |
| Get the Unicode category of a character | GetUnicodeCategory See also Rune.GetUnicodeCategory . |
| Determine whether a character is in a particular Unicode category such as digit, letter, punctuation, control character, and so on | IsControl , IsDigit , IsHighSurrogate , IsLetter , IsLetterOrDigit , IsLower , IsLowSurrogate , IsNumber , IsPunctuation , IsSeparator , IsSurrogate , IsSurrogatePair , IsSymbol , IsUpper , and IsWhiteSpace See also corresponding methods on the Rune type. |

| To do this | Use these <code>System.Char</code> methods |
|---|--|
| Convert a <code>Char</code> object that represents a number to a numeric value type | <code>GetNumericValue</code> See also Rune.GetNumericValue . |
| Convert a character in a string into a <code>Char</code> object | <code>Parse</code> and <code>TryParse</code> |
| Convert a <code>Char</code> object to a <code>String</code> object | <code>ToString</code> |
| Change the case of a <code>Char</code> object | <code>ToLower</code> , <code>ToLowerInvariant</code> , <code>ToUpper</code> , and <code>ToUpperInvariant</code> See also corresponding methods on the <code>Rune</code> type. |

Char values and interop

When a managed `Char` type, which is represented as a Unicode UTF-16 encoded code unit, is passed to unmanaged code, the interop marshaller converts the character set to ANSI by default. You can apply the `DllImportAttribute` attribute to platform invoke declarations and the `StructLayoutAttribute` attribute to a COM interop declaration to control which character set a marshaled `Char` type uses.

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



.NET feedback

.NET is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

System.StringComparer class

Article • 01/08/2024

This article provides supplementary remarks to the reference documentation for this API.

An object derived from the [StringComparer](#) class embodies string-based comparison, equality, and hash code operations that take into account both case and culture-specific comparison rules. You can use the [StringComparer](#) class to create a type-specific comparison to sort the elements in a generic collection. Classes such as [Hashtable](#), [Dictionary<TKey,TValue>](#), [SortedList](#), and [SortedList<TKey,TValue>](#) use the [StringComparer](#) class for sorting purposes.

A comparison operation that is represented by the [StringComparer](#) class is defined to be either case-sensitive or case-insensitive, and use either word (culture-sensitive) or ordinal (culture-insensitive) comparison rules. For more information about word and ordinal comparison rules, see [System.Globalization.CompareOptions](#).

ⓘ Note

You can download the [Default Unicode Collation Element Table](#), the latest version of the sort weight table. The specific version of the sort weight table depends on the version of the [International Components for Unicode](#) libraries installed on the system. For information on ICU versions and the Unicode versions that they implement, see [Downloading ICU](#).

For .NET Framework on Windows, you can download the [Sorting Weight Tables](#), a set of text files that contain information on the character weights used in sorting and comparison operations.

Implemented properties

You might be confused about how to use the [StringComparer](#) class properties because of a seeming contradiction. The [StringComparer](#) class is declared `abstract` (`MustInherit` in Visual Basic), which means its members can be invoked only on an object of a class derived from the [StringComparer](#) class. The contradiction is that each property of the [StringComparer](#) class is declared `static` (`Shared` in Visual Basic), which means the property can be invoked without first creating a derived class.

You can call a [StringComparer](#) property directly because each property actually returns an instance of an anonymous class that is derived from the [StringComparer](#) class. Consequently, the type of each property value is [StringComparer](#), which is the base class of the anonymous class, not the type of the anonymous class itself. Each [StringComparer](#) class property returns a [StringComparer](#) object that supports predefined case and comparison rules.

 [Collaborate with us on GitHub](#)

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

 [.NET feedback](#)

.NET is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

System.Text.Encoding class

Article • 01/08/2024

This article provides supplementary remarks to the reference documentation for this API.

The [Encoding](#) class represents a character encoding.

Encoding is the process of transforming a set of Unicode characters into a sequence of bytes. In contrast, decoding is the process of transforming a sequence of encoded bytes into a set of Unicode characters. For information about the Unicode Transformation Formats (UTFs) and other encodings supported by [Encoding](#), see [Character Encoding in .NET](#).

[Encoding](#) is intended to operate on Unicode characters instead of arbitrary binary data, such as byte arrays. If you must encode arbitrary binary data into text, you should use a protocol such as uuencode, which is implemented by methods such as [Convert.ToBase64CharArray](#).

.NET provides the following implementations of the [Encoding](#) class to support current Unicode encodings and other encodings:

- [ASCIIEncoding](#) encodes Unicode characters as single 7-bit ASCII characters. This encoding only supports character values between U+0000 and U+007F. Code page 20127. Also available through the [ASCII](#) property.
- [UTF7Encoding](#) encodes Unicode characters using the UTF-7 encoding. This encoding supports all Unicode character values. Code page 65000. Also available through the [UTF7](#) property.
- [UTF8Encoding](#) encodes Unicode characters using the UTF-8 encoding. This encoding supports all Unicode character values. Code page 65001. Also available through the [UTF8](#) property.
- [UnicodeEncoding](#) encodes Unicode characters using the UTF-16 encoding. Both little endian and big endian byte orders are supported. Also available through the [Unicode](#) property and the [BigEndianUnicode](#) property.
- [UTF32Encoding](#) encodes Unicode characters using the UTF-32 encoding. Both little endian (code page 12000) and big endian (code page 12001) byte orders are supported. Also available through the [UTF32](#) property.

The [Encoding](#) class is primarily intended to convert between different encodings and Unicode. Often one of the derived Unicode classes is the correct choice for your app.

Use the [GetEncoding](#) method to obtain other encodings, and call the [GetEncodings](#) method to get a list of all encodings.

List of encodings

The following table lists the encodings supported by .NET. It lists each encoding's code page number and the values of the encoding's [EncodingInfo.Name](#) and [EncodingInfo.DisplayName](#) properties. A check mark in the [.NET Framework support](#), [.NET Core support](#), or [.NET 5 and later support](#) column indicates that the code page is natively supported by that .NET implementation, regardless of the underlying platform. For .NET Framework, the availability of other encodings listed in the table depends on the operating system. For .NET Core and .NET 5 and later versions, other encodings are available by using the [System.Text.CodePagesEncodingProvider](#) class or by deriving from the [System.Text.EncodingProvider](#) class.

 **Note**

Code pages whose [EncodingInfo.Name](#) property corresponds to an international standard do not necessarily comply in full with that standard.

 Expand table

| Code page | Name | Display name | .NET Framework support | .NET Core support | .NET 5 and later support |
|-----------|----------|----------------------------|------------------------|-------------------|--------------------------|
| 37 | IBM037 | IBM EBCDIC (US-Canada) | | | |
| 437 | IBM437 | OEM United States | | | |
| 500 | IBM500 | IBM EBCDIC (International) | | | |
| 708 | ASMO-708 | Arabic (ASMO 708) | | | |
| 720 | DOS-720 | Arabic (DOS) | | | |
| 737 | ibm737 | Greek (DOS) | | | |
| 775 | ibm775 | Baltic (DOS) | | | |

| Code page | Name | Display name | .NET Framework support | .NET Core support | .NET 5 and later support |
|------------------|----------------|-----------------------------------|-------------------------------|--------------------------|---------------------------------|
| 850 | ibm850 | Western European (DOS) | | | |
| 852 | ibm852 | Central European (DOS) | | | |
| 855 | IBM855 | OEM Cyrillic | | | |
| 857 | ibm857 | Turkish (DOS) | | | |
| 858 | IBM00858 | OEM Multilingual Latin I | | | |
| 860 | IBM860 | Portuguese (DOS) | | | |
| 861 | ibm861 | Icelandic (DOS) | | | |
| 862 | DOS-862 | Hebrew (DOS) | | | |
| 863 | IBM863 | French Canadian (DOS) | | | |
| 864 | IBM864 | Arabic (864) | | | |
| 865 | IBM865 | Nordic (DOS) | | | |
| 866 | cp866 | Cyrillic (DOS) | | | |
| 869 | ibm869 | Greek, Modern (DOS) | | | |
| 870 | IBM870 | IBM EBCDIC (Multilingual Latin-2) | | | |
| 874 | windows-874 | Thai (Windows) | | | |
| 875 | cp875 | IBM EBCDIC (Greek Modern) | | | |
| 932 | shift_jis | Japanese (Shift-JIS) | | | |
| 936 | gb2312 | Chinese Simplified (GB2312) | ✓ | | |
| 949 | ks_c_5601-1987 | Korean | | | |
| 950 | big5 | Chinese Traditional | | | |

| Code page | Name | Display name | .NET Framework support | .NET Core support | .NET 5 and later support |
|------------------|--------------|----------------------------------|-------------------------------|--------------------------|---------------------------------|
| | | (Big5) | | | |
| 1026 | IBM1026 | IBM EBCDIC (Turkish Latin-5) | | | |
| 1047 | IBM01047 | IBM Latin-1 | | | |
| 1140 | IBM01140 | IBM EBCDIC (US-Canada-Euro) | | | |
| 1141 | IBM01141 | IBM EBCDIC (Germany-Euro) | | | |
| 1142 | IBM01142 | IBM EBCDIC (Denmark-Norway-Euro) | | | |
| 1143 | IBM01143 | IBM EBCDIC (Finland-Sweden-Euro) | | | |
| 1144 | IBM01144 | IBM EBCDIC (Italy-Euro) | | | |
| 1145 | IBM01145 | IBM EBCDIC (Spain-Euro) | | | |
| 1146 | IBM01146 | IBM EBCDIC (UK-Euro) | | | |
| 1147 | IBM01147 | IBM EBCDIC (France-Euro) | | | |
| 1148 | IBM01148 | IBM EBCDIC (International-Euro) | | | |
| 1149 | IBM01149 | IBM EBCDIC (Icelandic-Euro) | | | |
| 1200 | utf-16 | Unicode | ✓ | ✓ | ✓ |
| 1201 | unicodeFFE | Unicode (Big endian) | ✓ | ✓ | ✓ |
| 1250 | windows-1250 | Central European (Windows) | | | |
| 1251 | windows-1251 | Cyrillic (Windows) | | | |

| Code page | Name | Display name | .NET Framework support | .NET Core support | .NET 5 and later support |
|------------------|-------------------|----------------------------|-------------------------------|--------------------------|---------------------------------|
| 1252 | Windows-1252 | Western European (Windows) | ✓ | | |
| 1253 | windows-1253 | Greek (Windows) | | | |
| 1254 | windows-1254 | Turkish (Windows) | | | |
| 1255 | windows-1255 | Hebrew (Windows) | | | |
| 1256 | windows-1256 | Arabic (Windows) | | | |
| 1257 | windows-1257 | Baltic (Windows) | | | |
| 1258 | windows-1258 | Vietnamese (Windows) | | | |
| 1361 | Johab | Korean (Johab) | | | |
| 10000 | macintosh | Western European (Mac) | | | |
| 10001 | x-mac-japanese | Japanese (Mac) | | | |
| 10002 | x-mac-chinesetrad | Chinese Traditional (Mac) | | | |
| 10003 | x-mac-korean | Korean (Mac) | ✓ | | |
| 10004 | x-mac-arabic | Arabic (Mac) | | | |
| 10005 | x-mac-hebrew | Hebrew (Mac) | | | |
| 10006 | x-mac-greek | Greek (Mac) | | | |
| 10007 | x-mac-cyrillic | Cyrillic (Mac) | | | |
| 10008 | x-mac-chinesesimp | Chinese Simplified (Mac) | ✓ | | |
| 10010 | x-mac-romanian | Romanian (Mac) | | | |
| 10017 | x-mac-ukrainian | Ukrainian (Mac) | | | |
| 10021 | x-mac-thai | Thai (Mac) | | | |
| 10029 | x-mac-ce | Central European (Mac) | | | |
| 10079 | x-mac-icelandic | Icelandic (Mac) | | | |

| Code page | Name | Display name | .NET Framework support | .NET Core support | .NET 5 and later support |
|------------------|-----------------|-----------------------------|-------------------------------|--------------------------|---------------------------------|
| 10081 | x-mac-turkish | Turkish (Mac) | | | |
| 10082 | x-mac-croatian | Croatian (Mac) | | | |
| 12000 | utf-32 | Unicode (UTF-32) | ✓ | ✓ | ✓ |
| 12001 | utf-32BE | Unicode (UTF-32 Big endian) | ✓ | ✓ | ✓ |
| 20000 | x-Chinese-CNS | Chinese Traditional (CNS) | | | |
| 20001 | x-cp20001 | TCA Taiwan | | | |
| 20002 | x-Chinese-Eten | Chinese Traditional (Eten) | | | |
| 20003 | x-cp20003 | IBM5550 Taiwan | | | |
| 20004 | x-cp20004 | TeleText Taiwan | | | |
| 20005 | x-cp20005 | Wang Taiwan | | | |
| 20105 | x-IA5 | Western European (IA5) | | | |
| 20106 | x-IA5-German | German (IA5) | | | |
| 20107 | x-IA5-Swedish | Swedish (IA5) | | | |
| 20108 | x-IA5-Norwegian | Norwegian (IA5) | | | |
| 20127 | us-ascii | US-ASCII | ✓ | ✓ | ✓ |
| 20261 | x-cp20261 | T.61 | | | |
| 20269 | x-cp20269 | ISO-6937 | | | |
| 20273 | IBM273 | IBM EBCDIC (Germany) | | | |
| 20277 | IBM277 | IBM EBCDIC (Denmark-Norway) | | | |
| 20278 | IBM278 | IBM EBCDIC (Finland-Sweden) | | | |
| 20280 | IBM280 | IBM EBCDIC (Italy) | | | |

| Code page | Name | Display name | .NET Framework support | .NET Core support | .NET 5 and later support |
|------------------|-------------------------|--|-------------------------------|--------------------------|---------------------------------|
| 20284 | IBM284 | IBM EBCDIC (Spain) | | | |
| 20285 | IBM285 | IBM EBCDIC (UK) | | | |
| 20290 | IBM290 | IBM EBCDIC (Japanese katakana) | | | |
| 20297 | IBM297 | IBM EBCDIC (France) | | | |
| 20420 | IBM420 | IBM EBCDIC (Arabic) | | | |
| 20423 | IBM423 | IBM EBCDIC (Greek) | | | |
| 20424 | IBM424 | IBM EBCDIC (Hebrew) | | | |
| 20833 | x-EBCDIC-KoreanExtended | IBM EBCDIC (Korean Extended) | | | |
| 20838 | IBM-Thai | IBM EBCDIC (Thai) | | | |
| 20866 | koi8-r | Cyrillic (KOI8-R) | | | |
| 20871 | IBM871 | IBM EBCDIC (Icelandic) | | | |
| 20880 | IBM880 | IBM EBCDIC (Cyrillic Russian) | | | |
| 20905 | IBM905 | IBM EBCDIC (Turkish) | | | |
| 20924 | IBM00924 | IBM Latin-1 | | | |
| 20932 | EUC-JP | Japanese (JIS 0208-1990 and 0212-1990) | | | |
| 20936 | x-cp20936 | Chinese Simplified (GB2312-80) | ✓ | | |
| 20949 | x-cp20949 | Korean Wansung | ✓ | | |

| Code page | Name | Display name | .NET Framework support | .NET Core support | .NET 5 and later support |
|------------------|--------------|--|-------------------------------|--------------------------|---------------------------------|
| 21025 | cp1025 | IBM EBCDIC (Cyrillic Serbian-Bulgarian) | | | |
| 21866 | koi8-u | Cyrillic (KOI8-U) | | | |
| 28591 | iso-8859-1 | Western European (ISO) | ✓ | ✓ | ✓ |
| 28592 | iso-8859-2 | Central European (ISO) | | | |
| 28593 | iso-8859-3 | Latin 3 (ISO) | | | |
| 28594 | iso-8859-4 | Baltic (ISO) | | | |
| 28595 | iso-8859-5 | Cyrillic (ISO) | | | |
| 28596 | iso-8859-6 | Arabic (ISO) | | | |
| 28597 | iso-8859-7 | Greek (ISO) | | | |
| 28598 | iso-8859-8 | Hebrew (ISO-Visual) | ✓ | | |
| 28599 | iso-8859-9 | Turkish (ISO) | | | |
| 28603 | iso-8859-13 | Estonian (ISO) | | | |
| 28605 | iso-8859-15 | Latin 9 (ISO) | | | |
| 29001 | x-Europa | Europa | | | |
| 38598 | iso-8859-8-i | Hebrew (ISO-Logical) | ✓ | | |
| 50220 | iso-2022-jp | Japanese (JIS) | ✓ | | |
| 50221 | csISO2022JP | Japanese (JIS-Allow 1 byte Kana) | ✓ | | |
| 50222 | iso-2022-jp | Japanese (JIS-Allow 1 byte Kana - SO/SI) | ✓ | | |
| 50225 | iso-2022-kr | Korean (ISO) | ✓ | | |
| 50227 | x-cp50227 | Chinese Simplified | ✓ | | |

| Code page | Name | Display name | .NET Framework support | .NET Core support | .NET 5 and later support |
|------------------|-------------|------------------------------|-------------------------------|--------------------------|---------------------------------|
| (ISO-2022) | | | | | |
| 51932 | euc-jp | Japanese (EUC) | ✓ | | |
| 51936 | EUC-CN | Chinese Simplified (EUC) | ✓ | | |
| 51949 | euc-kr | Korean (EUC) | ✓ | | |
| 52936 | hz-gb-2312 | Chinese Simplified (HZ) | ✓ | | |
| 54936 | GB18030 | Chinese Simplified (GB18030) | ✓ | | |
| 57002 | x-iscii-de | ISCII Devanagari | ✓ | | |
| 57003 | x-iscii-be | ISCII Bengali | ✓ | | |
| 57004 | x-iscii-ta | ISCII Tamil | ✓ | | |
| 57005 | x-iscii-te | ISCII Telugu | ✓ | | |
| 57006 | x-iscii-as | ISCII Assamese | ✓ | | |
| 57007 | x-iscii-or | ISCII Oriya | ✓ | | |
| 57008 | x-iscii-ka | ISCII Kannada | ✓ | | |
| 57009 | x-iscii-ma | ISCII Malayalam | ✓ | | |
| 57010 | x-iscii-gu | ISCII Gujarati | ✓ | | |
| 57011 | x-iscii-pa | ISCII Punjabi | ✓ | | |
| 65000 | utf-7 | Unicode (UTF-7) | ✓ | ✓ | |
| 65001 | utf-8 | Unicode (UTF-8) | ✓ | ✓ | ✓ |

The following example calls the [GetEncoding\(Int32\)](#) and [GetEncoding\(String\)](#) methods to get the Greek (Windows) code page encoding. It compares the [Encoding](#) objects returned by the method calls to show that they are equal, and then maps displays the Unicode code point and the corresponding code page value for each character in the Greek alphabet.

C#

```

using System;
using System.Text;

public class Example
{
    public static void Main()
    {
        Encoding enc = Encoding.GetEncoding(1253);
        Encoding altEnc = Encoding.GetEncoding("windows-1253");
        Console.WriteLine("{0} = Code Page {1}: {2}", enc.EncodingName,
                           altEnc.CodePage, enc.Equals(altEnc));
        string greekAlphabet = "Α α Β β Γ γ Δ δ Ε ε Ζ ζ Η η " +
                               "Θ θ Ι ι Κ κ Λ λ Μ μ Ν ν Ξ ξ " +
                               "Ο ο Π π Ρ ρ Σ σ Σ σ Τ τ Υ υ " +
                               "Φ φ Χ χ Ψ ψ Ω ω";
        Console.OutputEncoding = Encoding.UTF8;
        byte[] bytes = enc.GetBytes(greekAlphabet);
        Console.WriteLine("{0,-12} {1,20} {2,20:X2}", "Character",
                           "Unicode Code Point", "Code Page 1253");
        for (int ctr = 0; ctr < bytes.Length; ctr++) {
            if (greekAlphabet[ctr].Equals(' '))
                continue;

            Console.WriteLine("{0,-12} {1,20} {2,20:X2}", greekAlphabet[ctr],
                             GetCodePoint(greekAlphabet[ctr]), bytes[ctr]);
        }
    }

    private static string GetCodePoint(char ch)
    {
        string retVal = "u+";
        byte[] bytes = Encoding.Unicode.GetBytes(ch.ToString());
        for (int ctr = bytes.Length - 1; ctr >= 0; ctr--)
            retVal += bytes[ctr].ToString("X2");

        return retVal;
    }
}

// The example displays the following output:
//      Character      Unicode Code Point      Code Page 1253
//      A                  u+0391                 C1
//      α                  u+03B1                 E1
//      Β                  u+0392                 C2
//      β                  u+03B2                 E2
//      Γ                  u+0393                 C3
//      γ                  u+03B3                 E3
//      Δ                  u+0394                 C4
//      δ                  u+03B4                 E4
//      Ε                  u+0395                 C5
//      ε                  u+03B5                 E5
//      Ζ                  u+0396                 C6
//      ζ                  u+03B6                 E6
//      Η                  u+0397                 C7
//      η                  u+03B7                 E7

```

| | | | |
|----|---|--------|----|
| // | Θ | u+0398 | C8 |
| // | θ | u+03B8 | E8 |
| // | Ι | u+0399 | C9 |
| // | ι | u+03B9 | E9 |
| // | Κ | u+039A | CA |
| // | κ | u+03BA | EA |
| // | Λ | u+039B | CB |
| // | λ | u+03BB | EB |
| // | Μ | u+039C | CC |
| // | μ | u+03BC | EC |
| // | Ν | u+039D | CD |
| // | ν | u+03BD | ED |
| // | Ξ | u+039E | CE |
| // | ξ | u+03BE | EE |
| // | Ο | u+039F | CF |
| // | ο | u+03BF | EF |
| // | Π | u+03A0 | D0 |
| // | π | u+03C0 | F0 |
| // | Ρ | u+03A1 | D1 |
| // | ρ | u+03C1 | F1 |
| // | Σ | u+03A3 | D3 |
| // | σ | u+03C3 | F3 |
| // | ς | u+03C2 | F2 |
| // | Τ | u+03A4 | D4 |
| // | τ | u+03C4 | F4 |
| // | Υ | u+03A5 | D5 |
| // | υ | u+03C5 | F5 |
| // | Φ | u+03A6 | D6 |
| // | φ | u+03C6 | F6 |
| // | Χ | u+03A7 | D7 |
| // | χ | u+03C7 | F7 |
| // | Ψ | u+03A8 | D8 |
| // | ψ | u+03C8 | F8 |
| // | Ω | u+03A9 | D9 |
| // | ω | u+03C9 | F9 |

If the data to be converted is available only in sequential blocks (such as data read from a stream) or if the amount of data is so large that it needs to be divided into smaller blocks, you should use the [Decoder](#) or the [Encoder](#) provided by the [GetDecoder](#) method or the [GetEncoder](#) method, respectively, of a derived class.

The UTF-16 and the UTF-32 encoders can use the big endian byte order (most significant byte first) or the little endian byte order (least significant byte first). For example, the Latin Capital Letter A (U+0041) is serialized as follows (in hexadecimal):

- UTF-16 big endian byte order: 00 41
- UTF-16 little endian byte order: 41 00
- UTF-32 big endian byte order: 00 00 00 41
- UTF-32 little endian byte order: 41 00 00 00

It is generally more efficient to store Unicode characters using the native byte order. For example, it is better to use the little endian byte order on little endian platforms, such as Intel computers.

The [GetPreamble](#) method retrieves an array of bytes that includes the byte order mark (BOM). If this byte array is prefixed to an encoded stream, it helps the decoder to identify the encoding format used.

For more information on byte order and the byte order mark, see The Unicode Standard at the [Unicode home page](#).

Note that the encoding classes allow errors to:

- Silently change to a "?" character.
- Use a "best fit" character.
- Change to an application-specific behavior through use of the [EncoderFallback](#) and [DecoderFallback](#) classes with the U+FFFD Unicode replacement character.

You should throw an exception on any data stream error. An app either uses a "throwonerror" flag when applicable or uses the [EncoderExceptionFallback](#) and [DecoderExceptionFallback](#) classes. Best fit fallback is often not recommended because it can cause data loss or confusion and is slower than simple character replacements. For ANSI encodings, the best fit behavior is the default.

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



.NET feedback

.NET is an open source project.
Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

System.Text.RegularExpressions.Regex class

Article • 01/08/2024

This article provides supplementary remarks to the reference documentation for this API.

The [Regex](#) class represents .NET's regular expression engine. It can be used to quickly parse large amounts of text to find specific character patterns; to extract, edit, replace, or delete text substrings; and to add the extracted strings to a collection to generate a report.

Note

If you want to validate a string by determining whether it conforms to a particular pattern, you can use the [System.Configuration.RegexStringValidator](#) class.

To use regular expressions, you define the pattern that you want to identify in a text stream by using the syntax documented in [Regular Expression Language - Quick Reference](#). Next, you can optionally instantiate a [Regex](#) object. Finally, you call a method that performs some operation, such as replacing text that matches the regular expression pattern, or identifying a pattern match.

Note

For some common regular expression patterns, see [Regular Expression Examples](#). There are also a number of online libraries of regular expression patterns, such as the one at [Regular-Expressions.info](#).

For more information about the regular expression language, see [Regular Expression Language - Quick Reference](#) or download and print one of these brochures:

[Quick Reference in Word \(.docx\) format](#)  [Quick Reference in PDF \(.pdf\) format](#) 

Regex vs. String methods

The [System.String](#) class includes several search and comparison methods that you can use to perform pattern matching with text. For example, the [String.Contains](#), [String.EndsWith](#), and [String.StartsWith](#) methods determine whether a string instance

contains a specified substring; and the [String.IndexOf](#), [String.IndexOfAny](#), [String.LastIndexOf](#), and [String.LastIndexOfAny](#) methods return the starting position of a specified substring in a string. Use the methods of the [System.String](#) class when you are searching for a specific string. Use the [Regex](#) class when you are searching for a specific pattern in a string. For more information and examples, see [.NET Regular Expressions](#).

Static vs. instance methods

After you define a regular expression pattern, you can provide it to the regular expression engine in either of two ways:

- By instantiating a [Regex](#) object that represents the regular expression. To do this, you pass the regular expression pattern to a [Regex](#) constructor. A [Regex](#) object is immutable; when you instantiate a [Regex](#) object with a regular expression, that object's regular expression cannot be changed.
- By supplying both the regular expression and the text to search to a `static` (`Shared` in Visual Basic) [Regex](#) method. This enables you to use a regular expression without explicitly creating a [Regex](#) object.

All [Regex](#) pattern identification methods include both static and instance overloads.

The regular expression engine must compile a particular pattern before the pattern can be used. Because [Regex](#) objects are immutable, this is a one-time procedure that occurs when a [Regex](#) class constructor or a static method is called. To eliminate the need to repeatedly compile a single regular expression, the regular expression engine caches the compiled regular expressions used in static method calls. As a result, regular expression pattern-matching methods offer comparable performance for static and instance methods. However, caching can adversely affect performance in the following two cases:

- When you use static method calls with a large number of regular expressions. By default, the regular expression engine caches the 15 most recently used static regular expressions. If your application uses more than 15 static regular expressions, some regular expressions must be recompiled. To prevent this recompilation, you can increase the [Regex.CacheSize](#) property.
- When you instantiate new [Regex](#) objects with regular expressions that have previously been compiled. For example, the following code defines a regular expression to locate duplicated words in a text stream. Although the example uses a single regular expression, it instantiates a new [Regex](#) object to process each line of text. This results in the recompilation of the regular expression with each iteration of the loop.

C#

```
StreamReader sr = new StreamReader(filename);
string input;
string pattern = @"\b(\w+)\s\1\b";
while (sr.Peek() >= 0)
{
    input = sr.ReadLine();
    Regex rgx = new Regex(pattern, RegexOptions.IgnoreCase);
    MatchCollection matches = rgx.Matches(input);
    if (matches.Count > 0)
    {
        Console.WriteLine("{0} ({1} matches):", input, matches.Count);
        foreach (Match match in matches)
            Console.WriteLine("    " + match.Value);
    }
}
sr.Close();
```

To prevent recompilation, you should instantiate a single [Regex](#) object that is accessible to all code that requires it, as shown in the following rewritten example.

C#

```
StreamReader sr = new StreamReader(filename);
string input;
string pattern = @"\b(\w+)\s\1\b";
Regex rgx = new Regex(pattern, RegexOptions.IgnoreCase);

while (sr.Peek() >= 0)
{
    input = sr.ReadLine();
    MatchCollection matches = rgx.Matches(input);
    if (matches.Count > 0)
    {
        Console.WriteLine("{0} ({1} matches):", input, matches.Count);
        foreach (Match match in matches)
            Console.WriteLine("    " + match.Value);
    }
}
sr.Close();
```

Perform regular expression operations

Whether you decide to instantiate a [Regex](#) object and call its methods or call static methods, the [Regex](#) class offers the following pattern-matching functionality:

- Validation of a match. You call the [IsMatch](#) method to determine whether a match is present.

- Retrieval of a single match. You call the [Match](#) method to retrieve a [Match](#) object that represents the first match in a string or in part of a string. Subsequent matches can be retrieved by calling the [Match.NextMatch](#) method.
- Retrieval of all matches. You call the [Matches](#) method to retrieve a [System.Text.RegularExpressions.MatchCollection](#) object that represents all the matches found in a string or in part of a string.
- Replacement of matched text. You call the [Replace](#) method to replace matched text. The replacement text can also be defined by a regular expression. In addition, some of the [Replace](#) methods include a [MatchEvaluator](#) parameter that enables you to programmatically define the replacement text.
- Creation of a string array that is formed from parts of an input string. You call the [Split](#) method to split an input string at positions that are defined by the regular expression.

In addition to its pattern-matching methods, the [Regex](#) class includes several special-purpose methods:

- The [Escape](#) method escapes any characters that may be interpreted as regular expression operators in a regular expression or input string.
- The [Unescape](#) method removes these escape characters.
- The [CompileToAssembly](#) method creates an assembly that contains predefined regular expressions. .NET contains examples of these special-purpose assemblies in the [System.Web.RegularExpressions](#) namespace.

Define a time-out value

.NET supports a full-featured regular expression language that provides substantial power and flexibility in pattern matching. However, the power and flexibility come at a cost: the risk of poor performance. Regular expressions that perform poorly are surprisingly easy to create. In some cases, regular expression operations that rely on excessive backtracking can appear to stop responding when they process text that nearly matches the regular expression pattern. For more information about the .NET Regular Expression engine, see [Details of Regular Expression Behavior](#). For more information about excessive backtracking, see [Backtracking](#).

Starting with .NET Framework 4.5, you can define a time-out interval for regular expression matches to limit excessive backtracking. Depending on the regular expression pattern and the input text, the execution time may exceed the specified

time-out interval, but it will not spend more time backtracking than the specified time-out interval. If the regular expression engine times out, it throws a [RegexMatchTimeoutException](#) exception. In most cases, this prevents the regular expression engine from wasting processing power by trying to match text that nearly matches the regular expression pattern. It also could indicate, however, that the time-out interval has been set too low, or that the current machine load has caused an overall degradation in performance.

How you handle the exception depends on the cause of the exception. If the exception occurs because the time-out interval is set too low or because of excessive machine load, you can increase the time-out interval and retry the matching operation. If the exception occurs because the regular expression relies on excessive backtracking, you can assume that a match does not exist, and, optionally, you can log information that will help you modify the regular expression pattern.

You can set a time-out interval by calling the [Regex\(String, RegexOptions, TimeSpan\)](#) constructor when you instantiate a regular expression object. For static methods, you can set a time-out interval by calling an overload of a matching method that has a `matchTimeout` parameter. If you do not set a time-out value explicitly, the default time-out value is determined as follows:

- By using the application-wide time-out value, if one exists. This can be any time-out value that applies to the application domain in which the [Regex](#) object is instantiated or the static method call is made. You can set the application-wide time-out value by calling the [AppDomain.SetData](#) method to assign the string representation of a [TimeSpan](#) value to the "REGEX_DEFAULT_MATCH_TIMEOUT" property.
- By using the value [InfiniteMatchTimeout](#), if no application-wide time-out value has been set.

Important

We recommend that you set a time-out value in all regular expression pattern-matching operations. For more information, see [Best Practices for Regular Expressions](#).

Examples

The following example uses a regular expression to check for repeated occurrences of words in a string. The regular expression `\b(?<word>\w+)\s+(\k<word>)\b` can be

interpreted as shown in the following table.

[\[\] Expand table](#)

| Pattern | Description |
|------------------|---|
| \b | Start the match at a word boundary. |
| (? <word>\w+) | Match one or more word characters up to a word boundary. Name this captured group <code>word</code> . |
| \s+ | Match one or more white-space characters. |
| (\k<word>) | Match the captured group that is named <code>word</code> . |
| \b | Match a word boundary. |

C#

```
using System;
using System.Text.RegularExpressions;

public class Test
{
    public static void Main ()
    {
        // Define a regular expression for repeated words.
        Regex rx = new Regex(@"\b(?<word>\w+)\s+(\k<word>)\b",
            RegexOptions.Compiled | RegexOptions.IgnoreCase);

        // Define a test string.
        string text = "The the quick brown fox  fox jumps over the lazy dog
dog.";

        // Find matches.
        MatchCollection matches = rx.Matches(text);

        // Report the number of matches found.
        Console.WriteLine("{0} matches found in:\n {1}",
            matches.Count,
            text);

        // Report on each match.
        foreach (Match match in matches)
        {
            GroupCollection groups = match.Groups;
            Console.WriteLine('{0}' repeated at positions {1} and {2},
                groups["word"].Value,
                groups[0].Index,
                groups[1].Index);
        }
    }
}
```

```

}

// The example produces the following output to the console:
//      3 matches found in:
//          The the quick brown fox fox jumps over the lazy dog dog.
//          'The' repeated at positions 0 and 4
//          'fox' repeated at positions 20 and 25
//          'dog' repeated at positions 49 and 53

```

The following example illustrates the use of a regular expression to check whether a string either represents a currency value or has the correct format to represent a currency value. In this case, the regular expression is built dynamically from the [NumberFormatInfo.CurrencyDecimalSeparator](#), [CurrencyDecimalDigits](#), [NumberFormatInfo.CurrencySymbol](#), [NumberFormatInfo.NegativeSign](#), and [NumberFormatInfo.PositiveSign](#) properties for the en-US culture. The resulting regular expression is `^\s*[+-]?\s?\$?\s?(\d*\.\?\d{2}){1}\$`. This regular expression can be interpreted as shown in the following table.

[+] Expand table

| Pattern | Description |
|--|--|
| <code>^</code> | Start at the beginning of the string. |
| <code>\s*</code> | Match zero or more white-space characters. |
| <code>[+-]?</code> | Match zero or one occurrence of either the positive sign or the negative sign. |
| <code>\s?</code> | Match zero or one white-space character. |
| <code>\\$?</code> | Match zero or one occurrence of the dollar sign. |
| <code>\s?</code> | Match zero or one white-space character. |
| <code>\d*</code> | Match zero or more decimal digits. |
| <code>\.?</code> | Match zero or one decimal point symbol. |
| <code>(\d{2})?</code> | Capturing group 1: Match two decimal digits zero or one time. |
| <code>(\d*\.\?(\d{2})?) {1}</code> | Match the pattern of integral and fractional digits separated by a decimal point symbol at least one time. |
| <code>\$</code> | Match the end of the string. |

In this case, the regular expression assumes that a valid currency string does not contain group separator symbols, and that it has either no fractional digits or the number of fractional digits defined by the specified culture's [CurrencyDecimalDigits](#) property.

C#

```
using System;
using System.Globalization;
using System.Text.RegularExpressions;

public class Example
{
    public static void Main()
    {
        // Get the en-US NumberFormatInfo object to build the regular
        // expression pattern dynamically.
        NumberFormatInfo nfi = CultureInfo.GetCultureInfo("en-
US").NumberFormat;

        // Define the regular expression pattern.
        string pattern;
        pattern = @"\s*[";
        // Get the positive and negative sign symbols.
        pattern += Regex.Escape(nfi.PositiveSign + nfi.NegativeSign) + @"]?";
        pattern += @"\s?"; // Get the currency symbol.
        pattern += Regex.Escape(nfi.CurrencySymbol) + @"?\s?"; // Add integral digits to the pattern.
        pattern += @"(\d*"; // Add the decimal separator.
        pattern += Regex.Escape(nfi.CurrencyDecimalSeparator) + "?"; // Add the fractional digits.
        pattern += @"(\d{"; // Determine the number of fractional digits in currency values.
        pattern += nfi.CurrencyDecimalDigits.ToString() + "})?)\{1}\$"; // Check each test string against the regular expression.
        foreach (string test in tests)
        {
            if (rgx.IsMatch(test))
                Console.WriteLine($"{test} is a currency value.");
            else
                Console.WriteLine($"{test} is not a currency value.");
        }
    }
}

// The example displays the following output:
//      Pattern is ^\s*[+-]?\s?\$?\s?(\d*\.\?(\d{2}))?\{1}\$
```

```
//      -42 is a currency value.  
//      19.99 is a currency value.  
//      0.001 is not a currency value.  
//      100 USD is not a currency value.  
//      .34 is a currency value.  
//      0.34 is a currency value.  
//      1,052.21 is not a currency value.  
//      $10.62 is a currency value.  
//      +1.43 is a currency value.  
//      -$0.23 is a currency value.
```

Because the regular expression in this example is built dynamically, you don't know at design time whether the currency symbol, decimal sign, or positive and negative signs of the specified culture (en-US in this example) might be misinterpreted by the regular expression engine as regular expression language operators. To prevent any misinterpretation, the example passes each dynamically generated string to the [Escape](#) method.

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

.NET is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

System.Text.Rune struct

Article • 01/08/2024

This article provides supplementary remarks to the reference documentation for this API.

A [Rune](#) instance represents a Unicode scalar value, which means any code point excluding the surrogate range (U+D800..U+DFFF). The type's constructors and conversion operators validate the input, so consumers can call the APIs assuming that the underlying [Rune](#) instance is well formed.

If you aren't familiar with the terms Unicode scalar value, code point, surrogate range, and well-formed, see [Introduction to character encoding in .NET](#).

When to use the Rune type

Consider using the `Rune` type if your code:

- Calls APIs that require Unicode scalar values
- Explicitly handles surrogate pairs

APIs that require Unicode scalar values

If your code iterates through the `char` instances in a `string` or a `ReadOnlySpan<char>`, some of the `char` methods won't work correctly on `char` instances that are in the surrogate range. For example, the following APIs require a scalar value `char` to work correctly:

- [Char.GetNumericValue](#)
- [Char.GetUnicodeCategory](#)
- [Char.IsDigit](#)
- [Char.IsLetter](#)
- [Char.IsLetterOrDigit](#)
- [Char.IsLower](#)
- [Char.IsNumber](#)
- [Char.IsPunctuation](#)
- [Char.IsSymbol](#)
- [Char.IsUpper](#)

The following example shows code that won't work correctly if any of the `char` instances are surrogate code points:

C#

```
// THE FOLLOWING METHOD SHOWS INCORRECT CODE.  
// DO NOT DO THIS IN A PRODUCTION APPLICATION.  
int CountLettersBadExample(string s)  
{  
    int letterCount = 0;  
  
    foreach (char ch in s)  
    {  
        if (char.IsLetter(ch))  
        { letterCount++; }  
    }  
  
    return letterCount;  
}
```

Here's equivalent code that works with a `ReadOnlySpan<char>`:

C#

```
// THE FOLLOWING METHOD SHOWS INCORRECT CODE.  
// DO NOT DO THIS IN A PRODUCTION APPLICATION.  
static int CountLettersBadExample(ReadOnlySpan<char> span)  
{  
    int letterCount = 0;  
  
    foreach (char ch in span)  
    {  
        if (char.IsLetter(ch))  
        { letterCount++; }  
    }  
  
    return letterCount;  
}
```

The preceding code works correctly with some languages such as English:

C#

```
CountLettersInString("Hello")  
// Returns 5
```

But it won't work correctly for languages outside the Basic Multilingual Plane, such as Osage:

C#

```
CountLettersInString("հԱՀԶՅԱ ՈԱ")
// Returns 0
```

The reason this method returns incorrect results for Osage text is that the `char` instances for Osage letters are surrogate code points. No single surrogate code point has enough information to determine if it's a letter.

If you change this code to use `Rune` instead of `char`, the method works correctly with code points outside the Basic Multilingual Plane:

C#

```
int CountLetters(string s)
{
    int letterCount = 0;

    foreach (Rune rune in s.EnumerateRunes())
    {
        if (Rune.IsLetter(rune))
        { letterCount++; }

    }

    return letterCount;
}
```

Here's equivalent code that works with a `ReadOnlySpan<char>`:

C#

```
static int CountLetters(ReadOnlySpan<char> span)
{
    int letterCount = 0;

    foreach (Rune rune in span.EnumerateRunes())
    {
        if (Rune.IsLetter(rune))
        { letterCount++; }

    }

    return letterCount;
}
```

The preceding code counts Osage letters correctly:

C#

```
CountLettersInString("հազար օպ")
```

```
// Returns 8
```

Code that explicitly handles surrogate pairs

Consider using the `Rune` type if your code calls APIs that explicitly operate on surrogate code points, such as the following methods:

- `Char.IsSurrogate`
- `Char.IsSurrogatePair`
- `Char.IsHighSurrogate`
- `Char.IsLowSurrogate`
- `Char.ConvertFromUtf32`
- `Char.ConvertToUtf32`

For example, the following method has special logic to deal with surrogate `char` pairs:

C#

```
static void ProcessStringUseChar(string s)
{
    Console.WriteLine("Using char");

    for (int i = 0; i < s.Length; i++)
    {
        if (!char.IsSurrogate(s[i]))
        {
            Console.WriteLine($"Code point: {(int)(s[i])}");
        }
        else if (i + 1 < s.Length && char.IsSurrogatePair(s[i], s[i + 1]))
        {
            int codePoint = char.ConvertToUtf32(s[i], s[i + 1]);
            Console.WriteLine($"Code point: {codePoint}");
            i++; // so that when the loop iterates it's actually +2
        }
        else
        {
            throw new Exception("String was not well-formed UTF-16.");
        }
    }
}
```

Such code is simpler if it uses `Rune`, as in the following example:

C#

```
static void ProcessStringUseRune(string s)
{
    Console.WriteLine("Using Rune");
```

```
for (int i = 0; i < s.Length;)
{
    if (!Rune.TryGetRuneAt(s, i, out Rune rune))
    {
        throw new Exception("String was not well-formed UTF-16.");
    }

    Console.WriteLine($"Code point: {rune.Value}");
    i += rune.Utf16SequenceLength; // increment the iterator by the
    number of chars in this Rune
}
}
```

When not to use `Rune`

You don't need to use the `Rune` type if your code:

- Looks for exact `char` matches
- Splits a string on a known char value

Using the `Rune` type may return incorrect results if your code:

- Counts the number of display characters in a `string`

Look for exact `char` matches

The following code iterates through a `string` looking for specific characters, returning the index of the first match. There's no need to change this code to use `Rune`, as the code is looking for characters that are represented by a single `char`.

C#

```
int GetIndexOfFirstAToZ(string s)
{
    for (int i = 0; i < s.Length; i++)
    {
        char thisChar = s[i];
        if ('A' <= thisChar && thisChar <= 'Z')
        {
            return i; // found a match
        }
    }

    return -1; // didn't find 'A' - 'Z' in the input string
}
```

Split a string on a known `char`

It's common to call `string.Split` and use delimiters such as `' '` (space) or `','` (comma), as in the following example:

C#

```
string inputString = "𠮷𠮷𠮷";
string[] splitOnSpace = inputString.Split(' ');
string[] splitOnComma = inputString.Split(',');
```

There is no need to use `Rune` here, because the code is looking for characters that are represented by a single `char`.

Count the number of display characters in a `string`

The number of `Rune` instances in a string might not match the number of user-perceivable characters shown when displaying the string.

Since `Rune` instances represent Unicode scalar values, components that follow the [Unicode text segmentation guidelines](#) can use `Rune` as a building block for counting display characters.

The `StringInfo` type can be used to count display characters, but it doesn't count correctly in all scenarios for .NET implementations other than .NET 5+.

For more information, see [Grapheme clusters](#).

How to instantiate a `Rune`

There are several ways to get a `Rune` instance. You can use a constructor to create a `Rune` directly from:

- A code point.

C#

```
Rune a = new Rune(0x0061); // LATIN SMALL LETTER A
Rune b = new Rune(0x10421); // DESERET CAPITAL LETTER ER
```

- A single `char`.

C#

```
Rune c = new Rune('a');
```

- A surrogate `char` pair.

C#

```
Rune d = new Rune('\ud83d', '\udd2e'); // U+1F52E CRYSTAL BALL
```

All of the constructors throw an `ArgumentException` if the input doesn't represent a valid Unicode scalar value.

There are `Rune.TryCreate` methods available for callers who don't want exceptions to be thrown on failure.

`Rune` instances can also be read from existing input sequences. For instance, given a `ReadOnlySpan<char>` that represents UTF-16 data, the `Rune.DecodeFromUtf16` method returns the first `Rune` instance at the beginning of the input span. The `Rune.DecodeFromUtf8` method operates similarly, accepting a `ReadOnlySpan<byte>` parameter that represents UTF-8 data. There are equivalent methods to read from the end of the span instead of the beginning of the span.

Query properties of a `Rune`

To get the integer code point value of a `Rune` instance, use the `Rune.Value` property.

C#

```
Rune rune = new Rune('\ud83d', '\udd2e'); // U+1F52E CRYSTAL BALL
int codePoint = rune.Value; // = 128302 decimal (= 0x1F52E)
```

Many of the static APIs available on the `char` type are also available on the `Rune` type. For instance, `Rune.IsWhiteSpace` and `Rune.GetUnicodeCategory` are equivalents to `Char.IsWhiteSpace` and `Char.GetUnicodeCategory` methods. The `Rune` methods correctly handle surrogate pairs.

The following example code takes a `ReadOnlySpan<char>` as input and trims from both the start and the end of the span every `Rune` that isn't a letter or a digit.

C#

```
static ReadOnlySpan<char> TrimNonLettersAndNonDigits(ReadOnlySpan<char>
span)
```

```

{
    // First, trim from the front.
    // If any Rune can't be decoded
    // (return value is anything other than "Done"),
    // or if the Rune is a letter or digit,
    // stop trimming from the front and
    // instead work from the end.
    while (Rune.DecodeFromUtf16(span, out Rune rune, out int charsConsumed)
== OperationStatus.Done)
    {
        if (Rune.IsLetterOrDigit(rune))
        { break; }
        span = span[charsConsumed..];
    }

    // Next, trim from the end.
    // If any Rune can't be decoded,
    // or if the Rune is a letter or digit,
    // break from the loop, and we're finished.
    while (Rune.DecodeLastFromUtf16(span, out Rune rune, out int
charsConsumed) == OperationStatus.Done)
    {
        if (Rune.IsLetterOrDigit(rune))
        { break; }
        span = span[..^charsConsumed];
    }

    return span;
}

```

There are some API differences between `char` and `Rune`. For example:

- There is no `Rune` equivalent to `Char.IsSurrogate(Char)`, since `Rune` instances by definition can never be surrogate code points.
- The `Rune.GetUnicodeCategory` doesn't always return the same result as `Char.GetUnicodeCategory`. It does return the same value as `CharUnicodeInfo.GetUnicodeCategory`. For more information, see the **Remarks** on `Char.GetUnicodeCategory`.

Convert a `Rune` to UTF-8 or UTF-16

Since a `Rune` is a Unicode scalar value, it can be converted to UTF-8, UTF-16, or UTF-32 encoding. The `Rune` type has built-in support for conversion to UTF-8 and UTF-16.

The `Rune.EncodeToUtf16` converts a `Rune` instance to `char` instances. To query the number of `char` instances that would result from converting a `Rune` instance to UTF-16, use the `Rune.Utf16SequenceLength` property. Similar methods exist for UTF-8 conversion.

The following example converts a `Rune` instance to a `char` array. The code assumes you have a `Rune` instance in the `rune` variable:

```
C#
```

```
char[] chars = new char[rune.Utf16SequenceLength];
int numCharsWritten = rune.EncodeToUtf16(chars);
```

Since a `string` is a sequence of UTF-16 chars, the following example also converts a `Rune` instance to UTF-16:

```
C#
```

```
string theString = rune.ToString();
```

The following example converts a `Rune` instance to a `UTF-8` byte array:

```
C#
```

```
byte[] bytes = new byte[rune.Utf8SequenceLength];
int numBytesWritten = rune.EncodeToUtf8(bytes);
```

The `Rune.EncodeToUtf16` and `Rune.EncodeToUtf8` methods return the actual number of elements written. They throw an exception if the destination buffer is too short to contain the result. There are non-throwing `TryEncodeToUtf8` and `TryEncodeToUtf16` methods as well for callers who want to avoid exceptions.

Rune in .NET vs. other languages

The term "rune" is not defined in the Unicode Standard. The term dates back to [the creation of UTF-8](#). Rob Pike and Ken Thompson were looking for a term to describe what would eventually become known as a code point. [They settled on the term "rune"](#), and Rob Pike's later influence over the Go programming language helped popularize the term.

However, the .NET `Rune` type is not the equivalent of the Go `rune` type. In Go, the `rune` type is an [alias for int32](#). A Go rune is intended to represent a Unicode code point, but it can be any 32-bit value, including surrogate code points and values that are not legal Unicode code points.

For similar types in other programming languages, see [Rust's primitive char type](#) or [Swift's Unicode.Scalar type](#), both of which represent Unicode scalar values. They

provide functionality similar to .NET's `Rune` type, and they disallow instantiation of values that are not legal Unicode scalar values.

 **Collaborate with us on GitHub**

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

.NET is an open source project.
Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

System.Text.StringBuilder class

Article • 01/08/2024

This article provides supplementary remarks to the reference documentation for this API.

The [StringBuilder](#) class represents a string-like object whose value is a mutable sequence of characters.

StringBuilder versus String type

Although [StringBuilder](#) and [String](#) both represent sequences of characters, they are implemented differently. [String](#) is an immutable type. That is, each operation that appears to modify a [String](#) object actually creates a new string.

For example, the call to the [String.Concat](#) method in the following C# example appears to change the value of a string variable named `value`. In fact, the [Concat](#) method returns a `value` object that has a different value and address from the `value` object that was passed to the method. Note that the example must be compiled using the `/unsafe` compiler option.

C#

```
using System;

public class Example7
{
    public unsafe static void Main()
    {
        string value = "This is the first sentence" + ".";
        fixed (char* start = value)
        {
            value = String.Concat(value, "This is the second sentence. ");
            fixed (char* current = value)
            {
                Console.WriteLine(start == current);
            }
        }
    }
} // The example displays the following output:
//      False
```

For routines that perform extensive string manipulation (such as apps that modify a string numerous times in a loop), modifying a string repeatedly can exact a significant

performance penalty. The alternative is to use [StringBuilder](#), which is a mutable string class. Mutability means that once an instance of the class has been created, it can be modified by appending, removing, replacing, or inserting characters. A [StringBuilder](#) object maintains a buffer to accommodate expansions to the string. New data is appended to the buffer if room is available; otherwise, a new, larger buffer is allocated, data from the original buffer is copied to the new buffer, and the new data is then appended to the new buffer.

Important

Although the [StringBuilder](#) class generally offers better performance than the [String](#) class, you should not automatically replace [String](#) with [StringBuilder](#) whenever you want to manipulate strings. Performance depends on the size of the string, the amount of memory to be allocated for the new string, the system on which your code is executing, and the type of operation. You should be prepared to test your code to determine whether [StringBuilder](#) actually offers a significant performance improvement.

Consider using the [String](#) class under these conditions:

- When the number of changes that your code will make to a string is small. In these cases, [StringBuilder](#) might offer negligible or no performance improvement over [String](#).
- When you are performing a fixed number of concatenation operations, particularly with string literals. In this case, the compiler might combine the concatenation operations into a single operation.
- When you have to perform extensive search operations while you are building your string. The [StringBuilder](#) class lacks search methods such as `IndexOf` or `StartsWith`. You'll have to convert the [StringBuilder](#) object to a [String](#) for these operations, and this can negate the performance benefit from using [StringBuilder](#). For more information, see the [Search the text in a StringBuilder object](#) section.

Consider using the [StringBuilder](#) class under these conditions:

- When you expect your code to make an unknown number of changes to a string at design time (for example, when you are using a loop to concatenate a random number of strings that contain user input).
- When you expect your code to make a significant number of changes to a string.

How StringBuilder works

The `StringBuilder.Length` property indicates the number of characters the `StringBuilder` object currently contains. If you add characters to the `StringBuilder` object, its length increases until it equals the size of the `StringBuilder.Capacity` property, which defines the number of characters that the object can contain. If the number of added characters causes the length of the `StringBuilder` object to exceed its current capacity, new memory is allocated, the value of the `Capacity` property is doubled, new characters are added to the `StringBuilder` object, and its `Length` property is adjusted. Additional memory for the `StringBuilder` object is allocated dynamically until it reaches the value defined by the `StringBuilder.MaxCapacity` property. When the maximum capacity is reached, no further memory can be allocated for the `StringBuilder` object, and trying to add characters or expand it beyond its maximum capacity throws either an `ArgumentOutOfRangeException` or an `OutOfMemoryException` exception.

The following example illustrates how a `StringBuilder` object allocates new memory and increases its capacity dynamically as the string assigned to the object expands. The code creates a `StringBuilder` object by calling its default (parameterless) constructor. The default capacity of this object is 16 characters, and its maximum capacity is more than 2 billion characters. Appending the string "This is a sentence." results in a new memory allocation because the string length (19 characters) exceeds the default capacity of the `StringBuilder` object. The capacity of the object doubles to 32 characters, the new string is added, and the length of the object now equals 19 characters. The code then appends the string "This is an additional sentence." to the value of the `StringBuilder` object 11 times. Whenever the append operation causes the length of the `StringBuilder` object to exceed its capacity, its existing capacity is doubled and the `Append` operation succeeds.

C#

```
using System;
using System.Reflection;
using System.Text;

public class Example4
{
    public static void Main()
    {
        StringBuilder sb = new StringBuilder();
        ShowSBInfo(sb);
        sb.Append("This is a sentence.");
        ShowSBInfo(sb);
        for (int ctr = 0; ctr <= 10; ctr++)
        {
            sb.Append("This is an additional sentence.");
            ShowSBInfo(sb);
        }
    }
}
```

```

    }

    private static void ShowSBInfo(StringBuilder sb)
    {
        foreach (var prop in sb.GetType().GetProperties())
        {
            if (prop.GetIndexParameters().Length == 0)
                Console.WriteLine("{0}: {1:N0}      ", prop.Name,
prop.GetValue(sb));
        }
        Console.WriteLine();
    }
}

// The example displays the following output:
// Capacity: 16      MaxCapacity: 2,147,483,647      Length: 0
// Capacity: 32      MaxCapacity: 2,147,483,647      Length: 19
// Capacity: 64      MaxCapacity: 2,147,483,647      Length: 50
// Capacity: 128     MaxCapacity: 2,147,483,647      Length: 81
// Capacity: 128     MaxCapacity: 2,147,483,647      Length: 112
// Capacity: 256     MaxCapacity: 2,147,483,647      Length: 143
// Capacity: 256     MaxCapacity: 2,147,483,647      Length: 174
// Capacity: 256     MaxCapacity: 2,147,483,647      Length: 205
// Capacity: 256     MaxCapacity: 2,147,483,647      Length: 236
// Capacity: 512     MaxCapacity: 2,147,483,647      Length: 267
// Capacity: 512     MaxCapacity: 2,147,483,647      Length: 298
// Capacity: 512     MaxCapacity: 2,147,483,647      Length: 329
// Capacity: 512     MaxCapacity: 2,147,483,647      Length: 360

```

Memory allocation

The default capacity of a [StringBuilder](#) object is 16 characters, and its default maximum capacity is [Int32.MaxValue](#). These default values are used if you call the [StringBuilder\(\)](#) and [StringBuilder\(String\)](#) constructors.

You can explicitly define the initial capacity of a [StringBuilder](#) object in the following ways:

- By calling any of the [StringBuilder](#) constructors that includes a `capacity` parameter when you create the object.
- By explicitly assigning a new value to the [StringBuilder.Capacity](#) property to expand an existing [StringBuilder](#) object. Note that the property throws an exception if the new capacity is less than the existing capacity or greater than the [StringBuilder](#) object's maximum capacity.
- By calling the [StringBuilder.EnsureCapacity](#) method with the new capacity. The new capacity must not be greater than the [StringBuilder](#) object's maximum capacity. However, unlike an assignment to the [Capacity](#) property, [EnsureCapacity](#) does not

throw an exception if the desired new capacity is less than the existing capacity; in this case, the method call has no effect.

If the length of the string assigned to the [StringBuilder](#) object in the constructor call exceeds either the default capacity or the specified capacity, the [Capacity](#) property is set to the length of the string specified with the `value` parameter.

You can explicitly define the maximum capacity of a [StringBuilder](#) object by calling the [StringBuilder\(Int32, Int32\)](#) constructor. You can't change the maximum capacity by assigning a new value to the [MaxCapacity](#) property, because it is read-only.

As the previous section shows, whenever the existing capacity is inadequate, additional memory is allocated and the capacity of a [StringBuilder](#) object doubles up to the value defined by the [MaxCapacity](#) property.

In general, the default capacity and maximum capacity are adequate for most apps. You might consider setting these values under the following conditions:

- If the eventual size of the [StringBuilder](#) object is likely to grow exceedingly large, typically in excess of several megabytes. In this case, there may be some performance benefit from setting the initial [Capacity](#) property to a significantly high value to eliminate the need for too many memory reallocations.
- If your code is running on a system with limited memory. In this case, you may want to consider setting the [MaxCapacity](#) property to less than [Int32.MaxValue](#) if your code is handling large strings that may cause it to execute in a memory-constrained environment.

Instantiate a [StringBuilder](#) object

You instantiate a [StringBuilder](#) object by calling one of its six overloaded class constructors, which are listed in the following table. Three of the constructors instantiate a [StringBuilder](#) object whose value is an empty string, but set its [Capacity](#) and [MaxCapacity](#) values differently. The remaining three constructors define a [StringBuilder](#) object that has a specific string value and capacity. Two of the three constructors use the default maximum capacity of [Int32.MaxValue](#), whereas the third allows you to set the maximum capacity.

[+] Expand table

| Constructor | String value | Capacity | Maximum capacity |
|-------------------------------------|--|--|--------------------------------------|
| StringBuilder() | String.Empty | 16 | Int32.MaxValue |
| StringBuilder(Int32) | String.Empty | Defined by the capacity parameter | Int32.MaxValue |
| StringBuilder(Int32, Int32) | String.Empty | Defined by the capacity parameter | Defined by the maxCapacity parameter |
| StringBuilder(String) | Defined by the value parameter | 16 or value.Length, whichever is greater | Int32.MaxValue |
| StringBuilder(String, Int32) | Defined by the value parameter | Defined by the capacity parameter or value.Length, whichever is greater. | Int32.MaxValue |
| StringBuilder(String, Int32, Int32) | Defined by value.Substring(startIndex, length) | Defined by the capacity parameter or value.Length, whichever is greater. | Int32.MaxValue |

The following example uses three of these constructor overloads to instantiate [StringBuilder](#) objects.

C#

```
using System;
using System.Text;

public class Example8
{
    public static void Main()
    {
        string value = "An ordinary string";
        int index = value.IndexOf("An ") + 3;
        int capacity = 0xFFFF;

        // Instantiate a StringBuilder from a string.
        StringBuilder sb1 = new StringBuilder(value);
        ShowSBInfo(sb1);

        // Instantiate a StringBuilder from string and define a capacity.
        StringBuilder sb2 = new StringBuilder(value, capacity);
        ShowSBInfo(sb2);

        // Instantiate a StringBuilder from substring and define a capacity.
        StringBuilder sb3 = new StringBuilder(value, index,
```

```

                value.Length - index,
                capacity);

        ShowSBInfo(sb3);
    }

    public static void ShowSBInfo(StringBuilder sb)
    {
        Console.WriteLine("\nValue: {0}", sb.ToString());
        foreach (var prop in sb.GetType().GetProperties())
        {
            if (prop.GetIndexParameters().Length == 0)
                Console.Write("{0}: {1:N0}      ", prop.Name,
prop.GetValue(sb));
            }
            Console.WriteLine();
        }
    }
    // The example displays the following output:
    //   Value: An ordinary string
    //   Capacity: 18      MaxCapacity: 2,147,483,647      Length: 18
    //
    //   Value: An ordinary string
    //   Capacity: 65,535     MaxCapacity: 2,147,483,647      Length: 18
    //
    //   Value: ordinary string
    //   Capacity: 65,535     MaxCapacity: 2,147,483,647      Length: 15

```

Call StringBuilder methods

Most of the methods that modify the string in a [StringBuilder](#) instance return a reference to that same instance. This enables you to call [StringBuilder](#) methods in two ways:

- You can make individual method calls and ignore the return value, as the following example does.

C#

```

using System;
using System.Text;

public class Example
{
    public static void Main()
    {
        StringBuilder sb = new StringBuilder();
        sb.Append("This is the beginning of a sentence, ");
        sb.Replace("the beginning of ", "");
        sb.Insert(sb.ToString().IndexOf("a ") + 2, "complete ");
        sb.Replace(",", ".");
        Console.WriteLine(sb.ToString());
    }
}

```

```
}
```

```
// The example displays the following output:
```

```
//      This is a complete sentence.
```

- You can make a series of method calls in a single statement. This can be convenient if you want to write a single statement that chains successive operations. The following example consolidates three method calls from the previous example into a single line of code.

C#

```
using System;
using System.Text;

public class Example2
{
    public static void Main()
    {
        StringBuilder sb = new StringBuilder("This is the beginning of
a sentence, ");
        sb.Replace("the beginning of ",
"").Insert(sb.ToString().IndexOf("a ") + 2,
"complete
").Replace(", ", ".");
        Console.WriteLine(sb.ToString());
    }
}

// The example displays the following output:
//      This is a complete sentence.
```

Perform `StringBuilder` operations

You can use the methods of the `StringBuilder` class to iterate, add, delete, or modify characters in a `StringBuilder` object.

Iterate `StringBuilder` characters

You can access the characters in a `StringBuilder` object by using the `StringBuilder.Chars[]` property. In C#, `Chars[]` is an indexer; in Visual Basic, it is the default property of the `StringBuilder` class. This enables you to set or retrieve individual characters by using their index only, without explicitly referencing the `Chars[]` property. Characters in a `StringBuilder` object begin at index 0 (zero) and continue to index `Length - 1`.

The following example illustrates the `Chars[]` property. It appends ten random numbers to a `StringBuilder` object, and then iterates each character. If the character's Unicode

category is `UnicodeCategory.DecimalDigitNumber`, it decreases the number by 1 (or changes the number to 9 if its value is 0). The example displays the contents of the `StringBuilder` object both before and after the values of individual characters were changed.

C#

```
using System;
using System.Globalization;
using System.Text;

public class Example3
{
    public static void Main()
    {
        Random rnd = new Random();
        StringBuilder sb = new StringBuilder();

        // Generate 10 random numbers and store them in a StringBuilder.
        for (int ctr = 0; ctr <= 9; ctr++)
            sb.Append(rnd.Next().ToString("N5"));

        Console.WriteLine("The original string:");
        Console.WriteLine(sb.ToString());

        // Decrease each number by one.
        for (int ctr = 0; ctr < sb.Length; ctr++)
        {
            if (Char.GetUnicodeCategory(sb[ctr]) ==
UnicodeCategory.DecimalDigitNumber)
            {
                int number = (int)Char.GetNumericValue(sb[ctr]);
                number--;
                if (number < 0) number = 9;

                sb[ctr] = number.ToString()[0];
            }
        }
        Console.WriteLine("\nThe new string:");
        Console.WriteLine(sb.ToString());
    }
}

// The example displays the following output:
//      The original string:
//
1,457,531,530.0000940,522,609.00001,668,113,564.00001,998,992,883.00001,
792,660,834.00
//
000101,203,251.00002,051,183,075.00002,066,000,067.00001,643,701,043.0000
01,702,382,508
//      .00000
//
//      The new string:
```

```
//  
0,346,420,429.99999839,411,598.999990,557,002,453.999990,887,881,772.999990,  
681,559,723.99  
//  
999090,192,140.999991,940,072,964.999991,955,999,956.999990,532,690,932.9999  
90,691,271,497  
// .99999
```

Using character-based indexing with the [Chars\[\]](#) property can be extremely slow under the following conditions:

- The [StringBuilder](#) instance is large (for example, it consists of several tens of thousands of characters).
- The [StringBuilder](#) is "chunky." That is, repeated calls to methods such as [StringBuilder.Append](#) have automatically expanded the object's [StringBuilder.Capacity](#) property and allocated new chunks of memory to it.

Performance is severely impacted because each character access walks the entire linked list of chunks to find the correct buffer to index into.

ⓘ Note

Even for a large "chunky" [StringBuilder](#) object, using the [Chars\[\]](#) property for index-based access to one or a small number of characters has a negligible performance impact; typically, it is an $O(n)$ operation. The significant performance impact occurs when iterating the characters in the [StringBuilder](#) object, which is an $O(n^2)$ operation.

If you encounter performance issues when using character-based indexing with [StringBuilder](#) objects, you can use any of the following workarounds:

- Convert the [StringBuilder](#) instance to a [String](#) by calling the [ToString](#) method, then access the characters in the string.
- Copy the contents of the existing [StringBuilder](#) object to a new pre-sized [StringBuilder](#) object. Performance improves because the new [StringBuilder](#) object is not chunky. For example:

C#

```
// sbOriginal is the existing StringBuilder object  
var sbNew = new StringBuilder(sbOriginal.ToString(),  
sbOriginal.Length);
```

- Set the initial capacity of the [StringBuilder](#) object to a value that is approximately equal to its maximum expected size by calling the [StringBuilder\(Int32\)](#) constructor. Note that this allocates the entire block of memory even if the [StringBuilder](#) rarely reaches its maximum capacity.

Add text to a [StringBuilder](#) object

The [StringBuilder](#) class includes the following methods for expanding the contents of a [StringBuilder](#) object:

- The [Append](#) method appends a string, a substring, a character array, a portion of a character array, a single character repeated multiple times, or the string representation of a primitive data type to a [StringBuilder](#) object.
- The [AppendLine](#) method appends a line terminator or a string along with a line terminator to a [StringBuilder](#) object.
- The [AppendFormat](#) method appends a [composite format string](#) to a [StringBuilder](#) object. The string representations of objects included in the result string can reflect the formatting conventions of the current system culture or a specified culture.
- The [Insert](#) method inserts a string, a substring, multiple repetitions of a string, a character array, a portion of a character array, or the string representation of a primitive data type at a specified position in the [StringBuilder](#) object. The position is defined by a zero-based index.

The following example uses the [Append](#), [AppendLine](#), [AppendFormat](#), and [Insert](#) methods to expand the text of a [StringBuilder](#) object.

C#

```
using System;
using System.Text;

public class Example6
{
    public static void Main()
    {
        // Create a StringBuilder object with no text.
        StringBuilder sb = new StringBuilder();
        // Append some text.
        sb.Append('*', 10).Append(" Adding Text to a StringBuilder Object ");
        sb.Append('*', 10);
        sb.AppendLine("\n");
        sb.AppendLine("Some code points and their corresponding
characters:");
        // Append some formatted text.
```

```

        for (int ctr = 50; ctr <= 60; ctr++)
    {
        sb.AppendFormat("{0,12:X4} {1,12}", ctr, Convert.ToChar(ctr));
        sb.AppendLine();
    }
    // Find the end of the introduction to the column.
    int pos = sb.ToString().IndexOf("characters:") + 11 +
        Environment.NewLine.Length;
    // Insert a column header.
    sb.Insert(pos, String.Format("{2}{0,12:X4} {1,12}{2}", "Code Unit",
        "Character", "\n"));

    // Convert the StringBuilder to a string and display it.
    Console.WriteLine(sb.ToString());
}
}

// The example displays the following output:
// **** Adding Text to a StringBuilder Object ****
//
// Some code points and their corresponding characters:
//
//      Code Unit      Character
//          0032          2
//          0033          3
//          0034          4
//          0035          5
//          0036          6
//          0037          7
//          0038          8
//          0039          9
//          003A          :
//          003B          ;
//          003C          <

```

Delete text from a StringBuilder object

The [StringBuilder](#) class includes methods that can reduce the size of the current [StringBuilder](#) instance. The [Clear](#) method removes all characters and sets the [Length](#) property to zero. The [Remove](#) method deletes a specified number of characters starting at a particular index position. In addition, you can remove characters from the end of a [StringBuilder](#) object by setting its [Length](#) property to a value that is less than the length of the current instance.

The following example removes some of the text from a [StringBuilder](#) object, displays its resulting capacity, maximum capacity, and length property values, and then calls the [Clear](#) method to remove all the characters from the [StringBuilder](#) object.

```

using System;
using System.Text;

public class Example5
{
    public static void Main()
    {
        StringBuilder sb = new StringBuilder("A StringBuilder object");
        ShowSBInfo(sb);
        // Remove "object" from the text.
        string textToRemove = "object";
        int pos = sb.ToString().IndexOf(textToRemove);
        if (pos >= 0)
        {
            sb.Remove(pos, textToRemove.Length);
            ShowSBInfo(sb);
        }
        // Clear the StringBuilder contents.
        sb.Clear();
        ShowSBInfo(sb);
    }

    public static void ShowSBInfo(StringBuilder sb)
    {
        Console.WriteLine("\nValue: {0}", sb.ToString());
        foreach (var prop in sb.GetType().GetProperties())
        {
            if (prop.GetIndexParameters().Length == 0)
                Console.Write("{0}: {1:N0}      ", prop.Name,
prop.GetValue(sb));
            }
            Console.WriteLine();
        }
    }
    // The example displays the following output:
    //   Value: A StringBuilder object
    //   Capacity: 22      MaxCapacity: 2,147,483,647      Length: 22
    //
    //   Value: A StringBuilder
    //   Capacity: 22      MaxCapacity: 2,147,483,647      Length: 16
    //
    //   Value:
    //   Capacity: 22      MaxCapacity: 2,147,483,647      Length: 0
}

```

Modify the text in a StringBuilder object

The [StringBuilder.Replace](#) method replaces all occurrences of a character or a string in the entire [StringBuilder](#) object or in a particular character range. The following example uses the [Replace](#) method to replace all exclamation points (!) with question marks (?) in the [StringBuilder](#) object.

C#

```
using System;
using System.Text;

public class Example13
{
    public static void Main()
    {
        StringBuilder MyStringBuilder = new StringBuilder("Hello World!");
        MyStringBuilder.Replace('!', '?');
        Console.WriteLine(MyStringBuilder);
    }
}
// The example displays the following output:
//      Hello World?
```

Search the text in a `StringBuilder` object

The `StringBuilder` class does not include methods similar to the `String.Contains`, `String.IndexOf`, and `String.StartsWith` methods provided by the `String` class, which allow you to search the object for a particular character or a substring. Determining the presence or starting character position of a substring requires that you search a `String` value by using either a string search method or a regular expression method. There are four ways to implement such searches, as the following table shows.

[] Expand table

| Technique | Pros | Cons |
|---|---|---|
| Search string values before adding them to the <code>StringBuilder</code> object. | Useful for determining whether a substring exists. | Cannot be used when the index position of a substring is important. |
| Call <code>ToString</code> and search the returned <code>String</code> object. | Easy to use if you assign all the text to a <code>StringBuilder</code> object, and then begin to modify it. | Cumbersome to repeatedly call <code>ToString</code> if you must make modifications before all text is added to the <code>StringBuilder</code> object. You must remember to work from the end of the <code>StringBuilder</code> object's text if you're making changes. |
| Use the <code>Chars[]</code> property to sequentially search a range of characters. | Useful if you're concerned with individual characters or a small substring. | Cumbersome if the number of characters to search is large or if the search logic is complex. |

| Technique | Pros | Cons |
|---|---|--|
| | | Results in very poor performance for objects that have grown very large through repeated method calls. |
| Convert the <code>StringBuilder</code> object to a <code>String</code> object, and perform modifications on the <code>String</code> object. | Useful if the number of modifications is small. | Negates the performance benefit of the <code>StringBuilder</code> class if the number of modifications is large. |

Let's examine these techniques in greater detail.

- If the goal of the search is to determine whether a particular substring exists (that is, if you aren't interested in the position of the substring), you can search strings before storing them in the `StringBuilder` object. The following example provides one possible implementation. It defines a `StringBuilderFinder` class whose constructor is passed a reference to a `StringBuilder` object and the substring to find in the string. In this case, the example tries to determine whether recorded temperatures are in Fahrenheit or Celsius, and adds the appropriate introductory text to the beginning of the `StringBuilder` object. A random number generator is used to select an array that contains data in either degrees Celsius or degrees Fahrenheit.

C#

```

using System;
using System.Text;

public class Example9
{
    public static void Main()
    {
        Random rnd = new Random();
        string[] tempF = { "47.6F", "51.3F", "49.5F", "62.3F" };
        string[] tempC = { "21.2C", "16.1C", "23.5C", "22.9C" };
        string[][] temps = { tempF, tempC };

        StringBuilder sb = new StringBuilder();
        var f = new StringBuilderFinder(sb, "F");
        var baseDate = new DateTime(2013, 5, 1);
        String[] temperatures = temps[rnd.Next(2)];
        bool isFahrenheit = false;
        foreach (var temperature in temperatures)
        {
            if (isFahrenheit)
                sb.AppendFormat("{0:d}: {1}\n", baseDate, temperature);
        }
    }
}

```

```

        else
            isFahrenheit = f.SearchAndAppend(String.Format("{0:d}:
{1}\n",
                                                baseDate,
temperature));
            baseDate = baseDate.AddDays(1);
        }
        if (isFahrenheit)
        {
            sb.Insert(0, "Average Daily Temperature in Degrees
Fahrenheit");
            sb.Insert(47, "\n\n");
        }
        else
        {
            sb.Insert(0, "Average Daily Temperature in Degrees
Celsius");
            sb.Insert(44, "\n\n");
        }
        Console.WriteLine(sb.ToString());
    }
}

public class StringBuilderFinder
{
    private StringBuilder sb;
    private String text;

    public StringBuilderFinder(StringBuilder sb, String textToFind)
    {
        this.sb = sb;
        this.text = textToFind;
    }

    public bool SearchAndAppend(String stringToSearch)
    {
        sb.Append(stringToSearch);
        return stringToSearch.Contains(text);
    }
}
// The example displays output similar to the following:
//      Average Daily Temperature in Degrees Celsius
//
//      5/1/2013: 21.2C
//      5/2/2013: 16.1C
//      5/3/2013: 23.5C
//      5/4/2013: 22.9C

```

- Call the `StringBuilder.ToString` method to convert the `StringBuilder` object to a `String` object. You can search the string by using methods such as `String.LastIndexOf` or `String.StartsWith`, or you can use regular expressions and the `Regex` class to search for patterns. Because both `StringBuilder` and `String` objects

use UTF-16 encoding to store characters, the index positions of characters, substrings, and regular expression matches are the same in both objects. This enables you to use [StringBuilder](#) methods to make changes at the same position at which that text is found in the [String](#) object.

ⓘ Note

If you adopt this approach, you should work from the end of the [StringBuilder](#) object to its beginning so that you don't have to repeatedly convert the [StringBuilder](#) object to a string.

The following example illustrates this approach. It stores four occurrences of each letter of the English alphabet in a [StringBuilder](#) object. It then converts the text to a [String](#) object and uses a regular expression to identify the starting position of each four-character sequence. Finally, it adds an underscore before each four-character sequence except for the first sequence, and converts the first character of the sequence to uppercase.

```
C#  
  
using System;  
using System.Text;  
using System.Text.RegularExpressions;  
  
public class Example10  
{  
    public static void Main()  
    {  
        // Create a StringBuilder object with 4 successive occurrences  
        // of each character in the English alphabet.  
        StringBuilder sb = new StringBuilder();  
        for (ushort ctr = (ushort)'a'; ctr <= (ushort)'z'; ctr++)  
            sb.Append(Convert.ToChar(ctr), 4);  
  
        // Create a parallel string object.  
        String sbString = sb.ToString();  
        // Determine where each new character sequence begins.  
        String pattern = @"(\w)\1+";  
        MatchCollection matches = Regex.Matches(sbString, pattern);  
  
        // Uppercase the first occurrence of the sequence, and separate  
        // it from the previous sequence by an underscore character.  
        for (int ctr = matches.Count - 1; ctr >= 0; ctr--)  
        {  
            Match m = matches[ctr];  
            sb[m.Index] = Char.ToUpper(sb[m.Index]);  
            if (m.Index > 0) sb.Insert(m.Index, "_");  
        }  
    }  
}
```

```

        }
        // Display the resulting string.
        sbString = sb.ToString();
        int line = 0;
        do
        {
            int nChars = line * 80 + 79 <= sbString.Length ?
                80 : sbString.Length - line * 80;
            Console.WriteLine(sbString.Substring(line * 80, nChars));
            line++;
        } while (line * 80 < sbString.Length);
    }
}

// The example displays the following output:
//
Aaaa_Bbbb_Cccc_Dddd_Eeee_Ffff_Gggg_Hhhh_Iiii_Jjjj_Kkkk_Llll_Mmmm_Nnnn_O
ooo_Pppp_
//    Qqqq_Rrrr_Ssss_Tttt_Uuuu_Vvvv_Wwww_Xxxx_Yyyy_Zzzz

```

- Use the [StringBuilder.Chars\[\]](#) property to sequentially search a range of characters in a [StringBuilder](#) object. This approach may not be practical if the number of characters to be searched is large or the search logic is particularly complex. For the performance implications of character-by-character index-based access for very large, chunked [StringBuilder](#) objects, see the documentation for the [StringBuilder.Chars\[\]](#) property.

The following example is identical in functionality to the previous example but differs in implementation. It uses the [Chars\[\]](#) property to detect when a character value has changed, inserts an underscore at that position, and converts the first character in the new sequence to uppercase.

C#

```

using System;
using System.Text;

public class Example11
{
    public static void Main()
    {
        // Create a StringBuilder object with 4 successive occurrences
        // of each character in the English alphabet.
        StringBuilder sb = new StringBuilder();
        for (ushort ctr = (ushort)'a'; ctr <= (ushort)'z'; ctr++)
            sb.Append(Convert.ToChar(ctr), 4);

        // Iterate the text to determine when a new character sequence
        // occurs.
        int position = 0;
        Char current = '\u0000';

```

```

do
{
    if (sb[position] != current)
    {
        current = sb[position];
        sb[position] = Char.ToUpper(sb[position]);
        if (position > 0)
            sb.Insert(position, "_");
        position += 2;
    }
    else
    {
        position++;
    }
} while (position <= sb.Length - 1);
// Display the resulting string.
String sbString = sb.ToString();
int line = 0;
do
{
    int nChars = line * 80 + 79 <= sbString.Length ?
                80 : sbString.Length - line * 80;
    Console.WriteLine(sbString.Substring(line * 80, nChars));
    line++;
} while (line * 80 < sbString.Length);
}
}

// The example displays the following output:
//
Aaaa_Bbbb_Cccc_Dddd_Eeee_Ffff_Gggg_Hhhh_Iiii_Jjjj_Kkkk_Llll_Mmmm_Nnnn_O
ooo_Pppp_
//    Qqqq_Rrrr_Ssss_Tttt_Uuuu_Vvvv_Wwww_Xxxx_Yyyy_Zzzz

```

- Store all the unmodified text in the [StringBuilder](#) object, call the [StringBuilder.ToString](#) method to convert the [StringBuilder](#) object to a [String](#) object, and perform the modifications on the [String](#) object. You can use this approach if you have only a few modifications; otherwise, the cost of working with immutable strings may negate the performance benefits of using a [StringBuilder](#) object.

The following example is identical in functionality to the previous two examples but differs in implementation. It creates a [StringBuilder](#) object, converts it to a [String](#) object, and then uses a regular expression to perform all remaining modifications on the string. The [Regex.Replace\(String, String, MatchEvaluator\)](#) method uses a lambda expression to perform the replacement on each match.

C#

```

using System;
using System.Text;

```

```

using System.Text.RegularExpressions;

public class Example12
{
    public static void Main()
    {
        // Create a StringBuilder object with 4 successive occurrences
        // of each character in the English alphabet.
        StringBuilder sb = new StringBuilder();
        for (ushort ctr = (ushort)'a'; ctr <= (ushort)'z'; ctr++)
            sb.Append(Convert.ToChar(ctr), 4);

        // Convert it to a string.
        String sbString = sb.ToString();

        // Use a regex to uppercase the first occurrence of the
        sequence,
        // and separate it from the previous sequence by an underscore.
        string pattern = @"(\w)(\1+)";
        sbString = Regex.Replace(sbString, pattern,
                                m => (m.Index > 0 ? "_" : "") +
                                m.Groups[1].Value.ToUpper() +
                                m.Groups[2].Value);

        // Display the resulting string.
        int line = 0;
        do
        {
            int nChars = line * 80 + 79 <= sbString.Length ?
                        80 : sbString.Length - line * 80;
            Console.WriteLine(sbString.Substring(line * 80, nChars));
            line++;
        } while (line * 80 < sbString.Length);
    }
}

// The example displays the following output:
//
Aaaa_Bbbb_Cccc_Dddd_Eeee_Ffff_Gggg_Hhhh_Iiii_Jjjj_Kkkk_Llll_Mmmm_Nnnn_O
ooo_Pppp_
//   Qqqq_Rrrr_Ssss_Tttt_Uuuu_Vvvv_Wwww_Xxxx_Yyyy_Zzzz

```

Convert the `StringBuilder` object to a `string`

You must convert the `StringBuilder` object to a `String` object before you can pass the string represented by the `StringBuilder` object to a method that has a `String` parameter or display it in the user interface. You perform this conversion by calling the `StringBuilder.ToString` method. For an illustration, see the previous example, which calls the `ToString` method to convert a `StringBuilder` object to a string so that it can be passed to a regular expression method.

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

.NET is an open source project.
Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

.NET regular expressions

Article • 03/08/2023

Regular expressions provide a powerful, flexible, and efficient method for processing text. The extensive pattern-matching notation of regular expressions enables you to quickly parse large amounts of text to:

- Find specific character patterns.
- Validate text to ensure that it matches a predefined pattern (such as an email address).
- Extract, edit, replace, or delete text substrings.
- Add extracted strings to a collection in order to generate a report.

For many applications that deal with strings or that parse large blocks of text, regular expressions are an indispensable tool.

How regular expressions work

The centerpiece of text processing with regular expressions is the regular expression engine, which is represented by the [System.Text.RegularExpressions.Regex](#) object in .NET. At a minimum, processing text using regular expressions requires that the regular expression engine be provided with the following two items of information:

- The regular expression pattern to identify in the text.

In .NET, regular expression patterns are defined by a special syntax or language, which is compatible with Perl 5 regular expressions and adds some additional features such as right-to-left matching. For more information, see [Regular Expression Language - Quick Reference](#).

- The text to parse for the regular expression pattern.

The methods of the [Regex](#) class let you perform the following operations:

- Determine whether the regular expression pattern occurs in the input text by calling the [Regex.IsMatch](#) method. For an example that uses the [IsMatch](#) method for validating text, see [How to: Verify that Strings Are in Valid Email Format](#).
- Retrieve one or all occurrences of text that matches the regular expression pattern by calling the [Regex.Match](#) or [Regex.Matches](#) method. The former method returns a [System.Text.RegularExpressions.Match](#) object that provides information about the matching text. The latter returns a [MatchCollection](#) object that contains one

`System.Text.RegularExpressions.Match` object for each match found in the parsed text.

- Replace text that matches the regular expression pattern by calling the `Regex.Replace` method. For examples that use the `Replace` method to change date formats and remove invalid characters from a string, see [How to: Strip Invalid Characters from a String](#) and [Example: Changing Date Formats](#).

For an overview of the regular expression object model, see [The Regular Expression Object Model](#).

For more information about the regular expression language, see [Regular Expression Language - Quick Reference](#) or download and print one of the following brochures:

- [Quick Reference in Word \(.docx\) format](#)
- [Quick Reference in PDF \(.pdf\) format](#)

Regular expression examples

The `String` class includes string search and replacement methods that you can use when you want to locate literal strings in a larger string. Regular expressions are most useful either when you want to locate one of several substrings in a larger string, or when you want to identify patterns in a string, as the following examples illustrate.

⚠ Warning

When using `System.Text.RegularExpressions` to process untrusted input, pass a timeout. A malicious user can provide input to `RegularExpression`, causing a [Denial-of-Service attack](#). ASP.NET Core framework APIs that use `RegularExpression` pass a timeout.

💡 Tip

The `System.Web.RegularExpressions` namespace contains a number of regular expression objects that implement predefined regular expression patterns for parsing strings from HTML, XML, and ASP.NET documents. For example, the `TagRegex` class identifies start tags in a string, and the `CommentRegex` class identifies ASP.NET comments in a string.

Example 1: Replace substrings

Assume that a mailing list contains names that sometimes include a title (Mr., Mrs., Miss, or Ms.) along with a first and last name. Suppose you don't want to include the titles when you generate envelope labels from the list. In that case, you can use a regular expression to remove the titles, as the following example illustrates:

```
C#  
  
using System;  
using System.Text.RegularExpressions;  
  
public class Example  
{  
    public static void Main()  
    {  
        string pattern = "(Mr\\..? |Mrs\\..? |Miss |Ms\\..? )";  
        string[] names = { "Mr. Henry Hunt", "Ms. Sara Samuels",  
                           "Abraham Adams", "Ms. Nicole Norris" };  
        foreach (string name in names)  
            Console.WriteLine(Regex.Replace(name, pattern, String.Empty));  
    }  
}  
// The example displays the following output:  
//     Henry Hunt  
//     Sara Samuels  
//     Abraham Adams  
//     Nicole Norris
```

The regular expression pattern `(Mr\..? |Mrs\..? |Miss |Ms\..?)` matches any occurrence of "Mr ", "Mr. ", "Mrs ", "Mrs. ", "Miss ", "Ms ", or "Ms. ". The call to the `Regex.Replace` method replaces the matched string with `String.Empty`; in other words, it removes it from the original string.

Example 2: Identify duplicated words

Accidentally duplicating words is a common error that writers make. Use a regular expression to identify duplicated words, as the following example shows:

```
C#  
  
using System;  
using System.Text.RegularExpressions;  
  
public class Class1  
{  
    public static void Main()  
    {  
        string pattern = @"\b(\w+)\s\1\b";  
        string input = "This this is a nice day. What about this? This tastes
```

```

good. I saw a a dog.";
    foreach (Match match in Regex.Matches(input, pattern,
RegexOptions.IgnoreCase))
        Console.WriteLine("{0} (duplicates '{1}') at position {2}",
                           match.Value, match.Groups[1].Value, match.Index);
    }
}
// The example displays the following output:
//      This this (duplicates 'This') at position 0
//      a a (duplicates 'a') at position 66

```

The regular expression pattern `\b(\w+?)\s\1\b` can be interpreted as follows:

| Pattern | Interpretation |
|---------------------|--|
| <code>\b</code> | Start at a word boundary. |
| <code>(\w+?)</code> | Match one or more word characters, but as few characters as possible. Together, they form a group that can be referred to as <code>\1</code> . |
| <code>\s</code> | Match a white-space character. |
| <code>\1</code> | Match the substring that's equal to the group named <code>\1</code> . |
| <code>\b</code> | Match a word boundary. |

The `Regex.Matches` method is called with regular expression options set to `RegexOptions.IgnoreCase`. Therefore, the match operation is case-insensitive, and the example identifies the substring "This this" as a duplication.

The input string includes the substring "this? This". However, because of the intervening punctuation mark, it isn't identified as a duplication.

Example 3: Dynamically build a culture-sensitive regular expression

The following example illustrates the power of regular expressions combined with the flexibility offered by .NET's globalization features. It uses the `NumberFormatInfo` object to determine the format of currency values in the system's current culture. It then uses that information to dynamically construct a regular expression that extracts currency values from the text. For each match, it extracts the subgroup that contains the numeric string only, converts it to a `Decimal` value, and calculates a running total.

C#

```

using System;
using System.Collections.Generic;

```

```

using System.Globalization;
using System.Text.RegularExpressions;

public class Example
{
    public static void Main()
    {
        // Define text to be parsed.
        string input = "Office expenses on 2/13/2008:\n" +
            "Paper (500 sheets)                $3.95\n" +
            "Pencils (box of 10)               $1.00\n" +
            "Pens (box of 10)                 $4.49\n" +
            "Erasers                          $2.19\n" +
            "Ink jet printer                  $69.95\n\n" +
            "Total Expenses                   $ 81.58\n";

        // Get current culture's NumberFormatInfo object.
        NumberFormatInfo nfi = CultureInfo.CurrentCulture.NumberFormat;
        // Assign needed property values to variables.
        string currencySymbol = nfi.CurrencySymbol;
        bool symbolPrecedesIfPositive = nfi.CurrencyPositivePattern % 2 == 0;
        string groupSeparator = nfi.CurrencyGroupSeparator;
        string decimalSeparator = nfi.CurrencyDecimalSeparator;

        // Form regular expression pattern.
        string pattern = Regex.Escape( symbolPrecedesIfPositive ?
currencySymbol : "") + @"^\s*[-+]?([0-9]{0,3})(\s+groupSeparator\s+|[0-
9]{3})*(\s+
Regex.Escape(decimalSeparator) + "[0-9]+)?)" +
(! symbolPrecedesIfPositive ? currencySymbol : ""));
        Console.WriteLine( "The regular expression pattern is:");
        Console.WriteLine("    " + pattern);

        // Get text that matches regular expression pattern.
        MatchCollection matches = Regex.Matches(input, pattern,
RegexOptions.IgnorePatternWhitespace);
        Console.WriteLine("Found {0} matches.", matches.Count);

        // Get numeric string, convert it to a value, and add it to List
object.
        List<decimal> expenses = new List<Decimal>();

        foreach (Match match in matches)
            expenses.Add(Decimal.Parse(match.Groups[1].Value));

        // Determine whether total is present and if present, whether it is
correct.
        decimal total = 0;
        foreach (decimal value in expenses)
            total += value;

        if (total / 2 == expenses[expenses.Count - 1])
            Console.WriteLine("The expenses total {0:C2}.",

```

```

expenses[expenses.Count - 1]);
else
    Console.WriteLine("The expenses total {0:C2}.", total);
}
}
// The example displays the following output:
//      The regular expression pattern is:
//      \$\s*[-+]?([0-9]{0,3}(,[0-9]{3})*(\.[0-9]+)?)?
//      Found 6 matches.
//      The expenses total $81.58.

```

On a computer whose current culture is English - United States (en-US), the example dynamically builds the regular expression `\$\s*[-+]?([0-9]{0,3}(,[0-9]{3})*(\.[0-9]+)?)?`. This regular expression pattern can be interpreted as follows:

| Pattern | Interpretation |
|---|---|
| <code>\\$</code> | Look for a single occurrence of the dollar symbol (\$) in the input string. The regular expression pattern string includes a backslash to indicate that the dollar symbol is to be interpreted literally rather than as a regular expression anchor. The \$ symbol alone would indicate that the regular expression engine should try to begin its match at the end of a string. To ensure that the current culture's currency symbol isn't misinterpreted as a regular expression symbol, the example calls the Regex.Escape method to escape the character. |
| <code>\s*</code> | Look for zero or more occurrences of a white-space character. |
| <code>[+-]?</code> | Look for zero or one occurrence of either a positive or negative sign. |
| <code>([0-9]{0,3}(,[0-9]{3})*(\.[0-9]+)?)?</code> | The outer parentheses define this expression as a capturing group or a subexpression. If a match is found, information about this part of the matching string can be retrieved from the second Group object in the GroupCollection object returned by the Match.Groups property. The first element in the collection represents the entire match. |
| <code>[0-9]{0,3}</code> | Look for zero to three occurrences of the decimal digits 0 through 9. |
| <code>(,[0-9]{3})*</code> | Look for zero or more occurrences of a group separator followed by three decimal digits. |
| <code>\.</code> | Look for a single occurrence of the decimal separator. |
| <code>[0-9]+</code> | Look for one or more decimal digits. |
| <code>(\.[0-9]+)?</code> | Look for zero or one occurrence of the decimal separator followed by at least one decimal digit. |

If each subpattern is found in the input string, the match succeeds, and a [Match](#) object that contains information about the match is added to the [MatchCollection](#) object.

Related articles

| Title | Description |
|---|--|
| Regular Expression Language - Quick Reference | Provides information on the set of characters, operators, and constructs that you can use to define regular expressions. |
| The Regular Expression Object Model | Provides information and code examples that illustrate how to use the regular expression classes. |
| Details of Regular Expression Behavior | Provides information about the capabilities and behavior of .NET regular expressions. |
| Use regular expressions in Visual Studio | |

Reference

- [System.Text.RegularExpressions](#)
- [System.Text.RegularExpressions.Regex](#)
- [Regular Expressions - Quick Reference \(download in Word format\)](#) ↗
- [Regular Expressions - Quick Reference \(download in PDF format\)](#) ↗

 Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

.NET regular expression source generators

Article • 06/01/2023

A regular expression, or regex, is a string that enables a developer to express a pattern being searched for, making it a very common way to search text and extract results as a subset from the searched string. In .NET, the `System.Text.RegularExpressions` namespace is used to define `Regex` instances and static methods, and match on user-defined patterns. In this article, you'll learn how to use source generation to generate `Regex` instances to optimize performance.

ⓘ Note

Where possible, use source generated regular expressions instead of compiling regular expressions using the `RegexOptions.Compiled` option. Source generation can help your app start faster, run more quickly and be more trimmable. To learn when source generation is possible, see [When to use it](#).

Compiled regular expressions

When you write `new Regex("somepattern")`, a few things happen. The specified pattern is parsed, both to ensure the validity of the pattern and to transform it into an internal tree that represents the parsed regex. The tree is then optimized in various ways, transforming the pattern into a functionally equivalent variation that can be more efficiently executed. The tree is written into a form that can be interpreted as a series of opcodes and operands that provide instructions to the regex interpreter engine on how to match. When a match is performed, the interpreter simply walks through those instructions, processing them against the input text. When instantiating a new `Regex` instance or calling one of the static methods on `Regex`, the interpreter is the default engine employed.

When you specify `RegexOptions.Compiled`, all of the same construction-time work would be performed. The resulting instructions would be transformed further by the reflection-emit-based compiler into IL instructions that would be written to a few `DynamicMethods`. When a match was performed, those `DynamicMethod`s would be invoked. This IL would essentially do exactly what the interpreter would do, except specialized for the exact pattern being processed. For example, if the pattern contained `[ac]`, the interpreter would see an opcode that said "match the input character at the

current position against the set specified in this set description" whereas the compiled IL would contain code that effectively said, "match the input character at the current position against 'a' or 'c'". This special casing and the ability to perform optimizations based on knowledge of the pattern are some of the main reasons for specifying `RegexOptions.Compiled` yields much faster-matching throughput than does the interpreter.

There are several downsides to `RegexOptions.Compiled`. The most impactful is that it incurs much more construction cost than using the interpreter. Not only are all of the same costs paid as for the interpreter, but it then needs to compile that resulting `RegexNode` tree and generated opcodes/operands into IL, which adds non-trivial expense. The generated IL further needs to be JIT-compiled on first use leading to even more expense at startup. `RegexOptions.Compiled` represents a fundamental tradeoff between overheads on the first use and overheads on every subsequent use. The use of `System.Reflection.Emit` also inhibits the use of `RegexOptions.Compiled` in certain environments; some operating systems don't permit dynamically generated code to be executed, and on such systems, `Compiled` will become a no-op.

Source generation

.NET 7 introduces a new `RegexGenerator` source generator. When the C# compiler was rewritten as the "[Roslyn](#)" C# compiler, it exposed object models for the entire compilation pipeline, as well as analyzers. More recently, Roslyn enabled source generators. Just like an analyzer, a source generator is a component that plugs into the compiler and is handed all of the same information as an analyzer, but in addition to being able to emit diagnostics, it can also augment the compilation unit with additional source code. The .NET 7 SDK includes a new source generator that recognizes the new `GeneratedRegexAttribute` on a partial method that returns `Regex`. The source generator provides an implementation of that method that implements all the logic for the `Regex`. For example, you might have written code like this:

C#

```
private static readonly Regex s_abcOrDefGeneratedRegex =
    new(pattern: "abc|def",
        options: RegexOptions.Compiled | RegexOptions.IgnoreCase);

private static void EvaluateText(string text)
{
    if (s_abcOrDefGeneratedRegex.IsMatch(text))
    {
        // Take action with matching text
    }
}
```

```
    }  
}
```

You can now rewrite the previous code as follows:

C#

```
[GeneratedRegex("abc|def", RegexOptions.IgnoreCase, "en-US")]  
private static partial Regex AbcOrDefGeneratedRegex();  
  
private static void EvaluateText(string text)  
{  
    if (AbcOrDefGeneratedRegex().IsMatch(text))  
    {  
        // Take action with matching text  
    }  
}
```

The generated implementation of `AbcOrDefGeneratedRegex()` similarly caches a singleton `Regex` instance, so no additional caching is needed to consume code.

💡 Tip

The `RegexOptions.Compiled` flag is ignored by the source generator, thus making it no longer needed in the source generated version.

```
/// <summary>Cached, thread-safe singleton instance.</summary>  
internal static readonly AbcOrDefGeneratedRegex_0 Instance = new AbcOrDefGeneratedRegex_0();
```

But as can be seen, it's not just doing `new Regex(...)`. Rather, the source generator is emitting as C# code a custom `Regex`-derived implementation with logic similar to what `RegexOptions.Compiled` emits in IL. You get all the throughput performance benefits of `RegexOptions.Compiled` (more, in fact) and the start-up benefits of `Regex.CompileToAssembly`, but without the complexity of `CompileToAssembly`. The source that's emitted is part of your project, which means it's also easily viewable and debuggable.

```

81     /// <summary>Search <paramref name="inputSpan"/> starting from base.runtexpos for the next location a
82     /// <param name="inputSpan">The text being scanned by the regular expression.</param>
83     /// <returns>true if a possible match was found; false if no more matches are possible.</returns>
84     private bool TryFindNextPossibleStartingPosition(ReadOnlySpan<char> inputSpan)
85     {
86         int pos = base.runtexpos;
87         ulong charMinusLow;
88
89         // Any possible match is at least 3 characters.
90         if (pos <= inputSpan.Length - 3)
91         {
92             // The pattern matches a character in the set [CFcf] at index 2.
93             // Find the next occurrence. If it can't be found, there's no match.
94             ReadOnlySpan<char> span = inputSpan.Slice(pos);
95             for (int i = 0; i < span.Length - 2; i++)
96             {
97                 int indexOfPos = span.Slice(i + 2).IndexOfAny("CFcf");
98                 if (indexOfPos < 0)
99                 {
100                     goto NoMatchFound;
101                 }
102                 i += indexOfPos;
103             }
104         }

```

Tip

In Visual Studio, right-click on your partial method declaration and select **Go To Definition**. Or, alternatively, select the project node in **Solution Explorer**, then expand **Dependencies > Analyzers > System.Text.RegularExpressions.Generator > System.Text.RegularExpressions.Generator.RegexGenerator > RegexGenerator.g.cs** to see the generated C# code from this regex generator.

You can set breakpoints in it, you can step through it, and you can use it as a learning tool to understand exactly how the regex engine is processing your pattern with your input. The generator even generates [triple-slash \(XML\) comments](#) to help make the expression understandable at a glance and where it's used.

```

[GeneratedRegex(pattern: "abc|def", options: RegexOptions.IgnoreCase)]
2 references | 0 changes | 0 authors, 0 changes
private static partial Regex AbcOrDefGeneratedRegex();

```

1 reference | 0 changes | 0 authors, 0 changes

```

private static void EvaluateText()
{
    if (s_abcOrDefGeneratedRegex
    {
        }
}

```

Regex Program.AbcOrDefGeneratedRegex()

Pattern explanation:

- o Match with 2 alternative expressions, atomically.
 - o Match a sequence of expressions.
 - o Match a character in the set [Aa].
 - o Match a character in the set [Bb].
 - o Match a character in the set [Cc].
 - o Match a sequence of expressions.
 - o Match a character in the set [Dd].
 - o Match a character in the set [Ee].
 - o Match a character in the set [Ff].

Inside the source-generated files

With .NET 7, both the source generator and `RegexCompiler` were almost entirely rewritten, fundamentally changing the structure of the generated code. This approach has been extended to handle all constructs (with one caveat), and both `RegexCompiler` and the source generator still map mostly 1:1 with each other, following the new approach. Consider the source generator output for one of the primary functions from the `(a|bc)d` expression:

C#

```
private bool TryMatchAtCurrentPosition(ReadOnlySpan<char> inputSpan)
{
    int pos = base.runtextrpos;
    int matchStart = pos;
    int capture_starting_pos = 0;
    ReadOnlySpan<char> slice = inputSpan.Slice(pos);

    // 1st capture group.
    //{
        capture_starting_pos = pos;

        // Match with 2 alternative expressions.
        //{
            if (slice.IsEmpty)
            {
                UncaptureUntil(0);
                return false; // The input didn't match.
            }

            switch (slice[0])
            {
                case 'a':
                    pos++;
                    slice = inputSpan.Slice(pos);
                    break;

                case 'b':
                    // Match 'c'.
                    if ((uint)slice.Length < 2 || slice[1] != 'c')
                    {
                        UncaptureUntil(0);
                        return false; // The input didn't match.
                    }

                    pos += 2;
                    slice = inputSpan.Slice(pos);
                    break;

                default:
                    UncaptureUntil(0);
                    return false; // The input didn't match.
            }
        //}
    //}
}
```

```

        base.Capture(1, capture_starting_pos, pos);
    //}

    // Match 'd'.
    if (slice.IsEmpty || slice[0] != 'd')
    {
        UncaptureUntil(0);
        return false; // The input didn't match.
    }

    // The input matched.
    pos++;
    base.runtextpos = pos;
    base.Capture(0, matchStart, pos);
    return true;

    // <summary>Undo captures until it reaches the specified capture
    position.</summary>
    [MethodImpl(MethodImplOptions.AggressiveInlining)]
    void UncaptureUntil(int capturePosition)
    {
        while (base.Crawlpos() > capturePosition)
        {
            base.Uncapture();
        }
    }
}

```

The goal of the source-generated code is to be understandable, with an easy-to-follow structure, with comments explaining what's being done at each step, and in general with code emitted under the guiding principle that the generator should emit code as if a human had written it. Even when backtracking is involved, the structure of the backtracking becomes part of the structure of the code, rather than relying on a stack to indicate where to jump next. For example, here's the code for the same generated matching function when the expression is `[ab]*[bc]`:

C#

```

private bool TryMatchAtCurrentPosition(ReadOnlySpan<char> inputSpan)
{
    int pos = base.runtextpos;
    int matchStart = pos;
    int charloop_starting_pos = 0, charloop_ending_pos = 0;
    ReadOnlySpan<char> slice = inputSpan.Slice(pos);

    // Match a character in the set [ab] greedily any number of times.
    //{
        charloop_starting_pos = pos;

        int iteration = slice.IndexOfAnyExcept('a', 'b');

```

```

    if (iteration < 0)
    {
        iteration = slice.Length;
    }

    slice = slice.Slice(iteration);
    pos += iteration;

    charloop_ending_pos = pos;
    goto CharLoopEnd;

CharLoopBacktrack:

    if (Utilities.s_hasTimeout)
    {
        base.CheckTimeout();
    }

    if (charloop_starting_pos >= charloop_ending_pos ||
        (charloop_ending_pos = inputSpan.Slice(
            charloop_starting_pos, charloop_ending_pos -
charloop_starting_pos
            .LastIndexOfAny('b', 'c')) < 0)
    {
        return false; // The input didn't match.
    }
    charloop_ending_pos += charloop_starting_pos;
    pos = charloop_ending_pos;
    slice = inputSpan.Slice(pos);

    CharLoopEnd:
}

// Advance the next matching position.
if (base.runtextrpos < pos)
{
    base.runtextrpos = pos;
}

// Match a character in the set [bc].
if (slice.IsEmpty || !char.IsBetween(slice[0], 'b', 'c'))
{
    goto CharLoopBacktrack;
}

// The input matched.
pos++;
base.runtextrpos = pos;
base.Capture(0, matchStart, pos);
return true;
}

```

You can see the structure of the backtracking in the code, with a `CharLoopBacktrack` label emitted for where to backtrack to and a `goto` used to jump to that location when a subsequent portion of the regex fails.

If you look at the code implementing `RegexCompiler` and the source generator, they will look extremely similar: similarly named methods, similar call structure, and even similar comments throughout the implementation. For the most part, they result in identical code, albeit one in IL and one in C#. Of course, the C# compiler is then responsible for translating the C# into IL, so the resulting IL in both cases likely won't be identical. The source generator relies on that in various cases, taking advantage of the fact that the C# compiler will further optimize various C# constructs. There are a few specific things the source generator will thus produce more optimized matching code than does `RegexCompiler`. For example, in one of the previous examples, you can see the source generator emitting a switch statement, with one branch for `'a'` and another branch for `'b'`. Because the C# compiler is very good at optimizing switch statements, with multiple strategies at its disposal for how to do so efficiently, the source generator has a special optimization that `RegexCompiler` does not. For `alternations`, the source generator looks at all of the branches, and if it can prove that every branch begins with a different starting character, it will emit a switch statement over that first character and avoid outputting any backtracking code for that alternation.

Here's a slightly more complicated example of that. In .NET 7, alternations are more heavily analyzed to determine whether it's possible to refactor them in a way that will make them more easily optimized by the backtracking engines and that will lead to simpler source-generated code. One such optimization supports extracting common prefixes from branches, and if the alternation is atomic such that ordering doesn't matter, reordering branches to allow for more such extraction. You can see the impact of that for the following weekday pattern

`Monday|Tuesday|Wednesday|Thursday|Friday|Saturday|Sunday`, which produces a matching function like this:

C#

```
private bool TryMatchAtCurrentPosition(ReadOnlySpan<char> inputSpan)
{
    int pos = base.runtextrpos;
    int matchStart = pos;
    ReadOnlySpan<char> slice = inputSpan.Slice(pos);

    // Match with 5 alternative expressions, atomically.
    {
        if (slice.IsEmpty)
        {
            return false; // The input didn't match.
        }
    }
}
```

```

    }

    switch (slice[0])
    {
        case 'M':
            // Match the string "onday".
            if (!slice.Slice(1).StartsWith("onday"))
            {
                return false; // The input didn't match.
            }

            pos += 6;
            slice = inputSpan.Slice(pos);
            break;

        case 'T':
            // Match with 2 alternative expressions, atomically.
            {
                if ((uint)slice.Length < 2)
                {
                    return false; // The input didn't match.
                }

                switch (slice[1])
                {
                    case 'u':
                        // Match the string "esday".
                        if (!slice.Slice(2).StartsWith("esday"))
                        {
                            return false; // The input didn't match.
                        }

                        pos += 7;
                        slice = inputSpan.Slice(pos);
                        break;

                    case 'h':
                        // Match the string "ursday".
                        if (!slice.Slice(2).StartsWith("ursday"))
                        {
                            return false; // The input didn't match.
                        }

                        pos += 8;
                        slice = inputSpan.Slice(pos);
                        break;

                    default:
                        return false; // The input didn't match.
                }
            }
    }

    break;

    case 'W':

```

```
// Match the string "ednesday".
if (!slice.Slice(1).StartsWith("ednesday"))
{
    return false; // The input didn't match.
}

pos += 9;
slice = inputSpan.Slice(pos);
break;

case 'F':
// Match the string "riday".
if (!slice.Slice(1).StartsWith("riday"))
{
    return false; // The input didn't match.
}

pos += 6;
slice = inputSpan.Slice(pos);
break;

case 'S':
// Match with 2 alternative expressions, atomically.
{
    if ((uint)slice.Length < 2)
    {
        return false; // The input didn't match.
    }

    switch (slice[1])
    {
        case 'a':
            // Match the string "turday".
            if (!slice.Slice(2).StartsWith("turday"))
            {
                return false; // The input didn't match.
            }

            pos += 8;
            slice = inputSpan.Slice(pos);
            break;

        case 'u':
            // Match the string "nday".
            if (!slice.Slice(2).StartsWith("nday"))
            {
                return false; // The input didn't match.
            }

            pos += 6;
            slice = inputSpan.Slice(pos);
            break;
    }
}

default:
return false; // The input didn't match.
```

```

        }

        break;

    default:
        return false; // The input didn't match.
    }
}

// The input matched.
base.runtextrpos = pos;
base.Capture(0, matchStart, pos);
return true;
}

```

Take notice of how `Thursday` was reordered to be just after `Tuesday`, and how for both the `Tuesday / Thursday` pair and the `Saturday / Sunday` pair, you end up with multiple levels of switches. In the extreme, if you were to create a long alternation of many different words, the source generator would end up emitting the logical equivalent of a trie¹, reading each character and `switch`'ing to the branch for handling the remainder of the word. This is a very efficient way to match words, and it's what the source generator is doing here.

At the same time, the source generator has other issues to contend with that simply don't exist when outputting to IL directly. If you look a couple of code examples back, you can see some braces somewhat strangely commented out. That's not a mistake. The source generator is recognizing that, if those braces weren't commented out, the structure of the backtracking is relying on jumping from outside of the scope to a label defined inside of that scope; such a label would not be visible to such a `goto` and the code would fail to compile. Thus, the source generator needs to avoid there being a scope in the way. In some cases, it'll simply comment out the scope as was done here. In other cases where that's not possible, it may sometimes avoid constructs that require scopes (such as a multi-statement `if` block) if doing so would be problematic.

The source generator handles everything `RegexCompiler` handles, with one exception. As with handling `RegexOptions.IgnoreCase`, the implementations now use a casing table to generate sets at construction time, and how `IgnoreCase` backreference matching needs to consult that casing table. That table is internal to

`System.Text.RegularExpressions.dll`, and for now, at least, the code external to that assembly (including code emitted by the source generator) does not have access to it. That makes handling `IgnoreCase` backreferences a challenge in the source generator and they aren't supported. This is the one construct not supported by the source generator that is supported by `RegexCompiler`. If you try to use a pattern that has one of

these (which is rare), the source generator won't emit a custom implementation and will instead fall back to caching a regular `Regex` instance:

```
/// <summary>Cached, thread-safe singleton instance.</summary>
internal static readonly Regex Instance =
    new Regex(pattern: "(\\w)\\1", options: RegexOptions.IgnoreCase)
```

Also, neither `RegexCompiler` nor the source generator supports the new `RegexOptions.NonBacktracking`. If you specify `RegexOptions.Compiled` | `RegexOptions.NonBacktracking`, the `Compiled` flag will just be ignored, and if you specify `NonBacktracking` to the source generator, it will similarly fall back to caching a regular `Regex` instance.

When to use it

The general guidance is if you can use the source generator, use it. If you're using `Regex` today in C# with arguments known at compile time, and especially if you're already using `RegexOptions.Compiled` (because the regex has been identified as a hot spot that would benefit from faster throughput), you should prefer to use the source generator. The source generator will give your regex the following benefits:

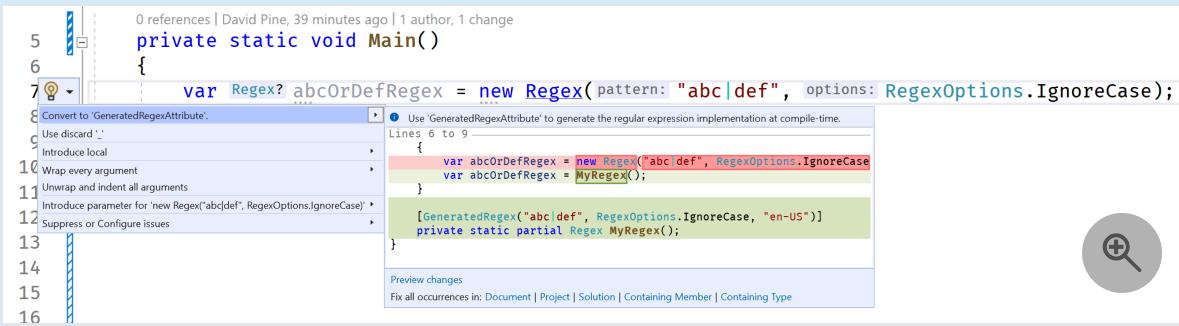
- All the throughput benefits of `RegexOptions.Compiled`.
- The startup benefits of not having to do all the regex parsing, analysis, and compilation at run time.
- The option of using ahead-of-time compilation with the code generated for the regex.
- Better debuggability and understanding of the regex.
- The possibility to reduce the size of your trimmed app by trimming out large swaths of code associated with `RegexCompiler` (and potentially even reflection emit itself).

When used with an option like `RegexOptions.NonBacktracking` for which the source generator can't generate a custom implementation, it will still emit caching and XML comments that describe the implementation, making it valuable. The main downside of the source generator is that it emits additional code into your assembly, so there's the potential for increased size. The more regexes in your app and the larger they are, the more code will be emitted for them. In some situations, just as `RegexOptions.Compiled` may be unnecessary, so too may be the source generator. For example, if you have a

regex that's needed only rarely and for which throughput doesn't matter, it could be more beneficial to just rely on the interpreter for that sporadic usage.

ⓘ Important

.NET 7 includes an [analyzer](#) that identifies the use of `Regex` that could be converted to the source generator, and a fixer that does the conversion for you:



See also

- SYSLIB diagnostics for regex source generation
- .NET regular expressions
- Backtracking in regular expressions
- Compilation and reuse in regular expressions
- Source generators
- .NET Blog: Regular Expression improvements in .NET 7 ↗

ⓘ Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



.NET feedback

The .NET documentation is open source. Provide feedback here.

ⓘ Open a documentation issue

ⓘ Provide product feedback

Regular Expression Language - Quick Reference

Article • 06/18/2022

A regular expression is a pattern that the regular expression engine attempts to match in input text. A pattern consists of one or more character literals, operators, or constructs. For a brief introduction, see [.NET Regular Expressions](#).

Each section in this quick reference lists a particular category of characters, operators, and constructs that you can use to define regular expressions.

We've also provided this information in two formats that you can download and print for easy reference:

- [Download in Word \(.docx\) format](#)
- [Download in PDF \(.pdf\) format](#)

Character Escapes

The backslash character (\) in a regular expression indicates that the character that follows it either is a special character (as shown in the following table), or should be interpreted literally. For more information, see [Character Escapes](#).

| Escaped character | Description | Pattern | Matches |
|-------------------|---|-----------|--|
| \a | Matches a bell character, \u0007. | \a | "\u0007" in "Error!" + '\u0007' |
| \b | In a character class, matches a backspace, \u0008. | [\b]{3,} | "\b\b\b\b" in "\b\b\b\b" |
| \t | Matches a tab, \u0009. | (\w+)\t | "item1\t", "item2\t" in "item1\item2\t" |
| \r | Matches a carriage return, \u000D. (\r is not equivalent to the newline character, \n.) | \r\n(\w+) | "\r\nThese" in "\r\nThese are\ntwo lines." |
| \v | Matches a vertical tab, \u000B. | [\v]{2,} | "\v\v\v" in "\v\v\v" |

| Escaped character | Description | Pattern | Matches |
|-------------------|---|------------------|---|
| \f | Matches a form feed, \u000C. | [\f]{2,} | "\f\f\f" in "\f\f\f" |
| \n | Matches a new line, \u000A. | \r\n(\w+) | "\r\nThese" in "\r\nThese are\n two lines." |
| \e | Matches an escape, \u001B. | \e | "\x001B" in "\x001B" |
| \ nnn | Uses octal representation to specify a character (<i>nnn</i> consists of two or three digits). | \w\040\w | "a b", "c d" in "a bc d" |
| \x nn | Uses hexadecimal representation to specify a character (<i>nn</i> consists of exactly two digits). | \w\x20\w | "a b", "c d" in "a bc d" |
| \c X | Matches the ASCII control character that is specified by <i>X</i> or <i>x</i> , where <i>X</i> or <i>x</i> is the letter of the control character. | \cc | "\x0003" in "\x0003" (Ctrl-C) |
| \c x | | | |
| \u nnnn | Matches a Unicode character by using hexadecimal representation (exactly four digits, as represented by <i>nnnn</i>). | \w\u0020\w | "a b", "c d" in "a bc d" |
| \ | When followed by a character that is not recognized as an escaped character in this and other tables in this topic, matches that character. For example, * is the same as \xA, and \. is the same as \xE. This allows the regular expression engine to disambiguate language elements (such as * or ?) and character literals (represented by * or \?). | \d+[\+\-\x*]\d+ | "2+2" and "3*9" in "(2+2) * 3*9" |

Character Classes

A character class matches any one of a set of characters. Character classes include the language elements listed in the following table. For more information, see [Character Classes](#).

| Character class | Description | Pattern | Matches |
|----------------------------|---|---------|---------------|
| [<i>character_group</i>] | Matches any single character in <i>character_group</i> . By default, the match is case-sensitive. | [ae] | "a" in "gray" |

| Character class | Description | Pattern | Matches |
|----------------------|---|----------------|--|
| | | | "a", "e" in "lane" |
| [^ character_group] | Negation: Matches any single character that is not in <i>character_group</i> . By default, characters in <i>character_group</i> are case-sensitive. | [^aei] | "r", "g", "n" in "reign" |
| [first - last] | Character range: Matches any single character in the range from <i>first</i> to <i>last</i> . | [A-Z] | "A", "B" in "AB123" |
| . | Wildcard: Matches any single character except \n . | a.e | "ave" in "nave" |
| | To match a literal period character (.) or (\u002E), you must precede it with the escape character (\.). | | "ate" in "water" |
| \p{ name } | Matches any single character in the Unicode general category or named block specified by <i>name</i> . | \p{Lu} | "C", "L" in "City Lights" |
| | | \p{IsCyrillic} | "д", "ж" in "Джем" |
| \P{ name } | Matches any single character that is not in the Unicode general category or named block specified by <i>name</i> . | \P{Lu} | "i", "t", "у" in "City" |
| | | \P{IsCyrillic} | "е", "м" in "Джем" |
| \w | Matches any word character . | \w | "I", "D", "A", "1", "3" in "ID A1.3" |
| \W | Matches any non-word character . | \W | " ", "." in "ID A1.3" |
| \s | Matches any white-space character . | \w\s | "D " in "ID A1.3" |
| \S | Matches any non-white-space character . | \s\S | " _" in "int _ctr" |
| \d | Matches any decimal digit . | \d | "4" in "4 = IV" |
| \D | Matches any character other than a decimal digit . | \D | " ", "=" , " " , "I", "v" in "4 |

| Character class | Description | Pattern | Matches |
|-----------------|-------------|---------|---------|
| | | = IV" | |

Anchors

Anchors, or atomic zero-width assertions, cause a match to succeed or fail depending on the current position in the string, but they do not cause the engine to advance through the string or consume characters. The metacharacters listed in the following table are anchors. For more information, see [Anchors](#).

| Assertion | Description | Pattern | Matches |
|-----------|---|--------------|---|
| ^ | By default, the match must start at the beginning of the string; in multiline mode, it must start at the beginning of the line. | ^\d{3} | "901" in "901-333-" |
| \$ | By default, the match must occur at the end of the string or before \n at the end of the string; in multiline mode, it must occur before the end of the line or before \n at the end of the line. | -\d{3}\$ | "-333" in "-901-333" |
| \A | The match must occur at the start of the string. | \A\d{3} | "901" in "901-333-" |
| \z | The match must occur at the end of the string or before \n at the end of the string. | -\d{3}\z | "-333" in "-901-333" |
| \z | The match must occur at the end of the string. | -\d{3}\z | "-333" in "-901-333" |
| \G | The match must occur at the point where the previous match ended, or if there was no previous match, at the position in the string where matching started. | \G\(\d\) | "(1)", "(3)", "(5)" in "(1)(3)(5)[7](9)" |
| \b | The match must occur on a boundary between a \w (alphanumeric) and a \W (nonalphanumeric) character. | \b\w+\s\w+\b | "them theme", "them them" in "them theme them them" |
| \B | The match must not occur on a \b boundary. | \Bend\w*\b | "ends", "ender" in "end sends endure lender" |

Grouping Constructs

Grouping constructs delineate subexpressions of a regular expression and typically capture substrings of an input string. Grouping constructs include the language elements listed in the following table. For more information, see [Grouping Constructs](#).

| Grouping construct | Description | Pattern | Matches |
|---|---|--|---|
| (<i>subexpression</i>) | Captures the matched subexpression and assigns it a one-based ordinal number. | (\w)\1 | "ee" in "deep" |
| (? <i>name</i> < <i>subexpression</i> >) or (?' <i>name</i> ' <i>subexpression</i>) | Captures the matched subexpression into a named group. | (?<double>\w)\k<double> | "ee" in "deep" |
| (? <i>name1</i> - <i>name2</i> > <i>subexpression</i>) or (?' <i>name1</i> - <i>name2</i> ' <i>subexpression</i>) | Defines a balancing group definition. For more information, see the "Balancing Group Definition" section in Grouping Constructs . | ((('Open'\()[^\\(\)]*+((('Close-Open'\())[^\(\)]*+)*((('Open')(!))\$ | "((1-3)*(3-1))" in "3+2*((1-3)*(3-1))" |
| (?: <i>subexpression</i>) | Defines a noncapturing group. | Write(?:Line)? | "WriteLine" in "Console.WriteLine()" "Write" in "Console.WriteLine(value)" |
| (?imnsx- imnsx: <i>subexpression</i>) | Applies or disables the specified options within <i>subexpression</i> . For more information, see Regular Expression Options . | A\d{2}(?i:\w+)\b | "A12x1", "A12XL" in "A12x1 A12XL a12x1" |
| (?= <i>subexpression</i>) | Zero-width positive lookahead assertion. | \b\w+\b(?.+and.+) | "cats", "dogs" in |

| Grouping construct | Description | Pattern | Matches |
|----------------------------|---|---|--|
| | | | "cats, dogs and some mice." |
| (?! subexpression) | Zero-width negative lookahead assertion. | \b\w+\b(?! .+and.+) | "and", "some", "mice" in "cats, dogs and some mice." |
| (?<= subexpression) | Zero-width positive lookbehind assertion. | \b\w+\b(?<=.+and.+) _____ \b\w+\b(?<=.+and.*) | "some", "mice" in "cats, dogs and some mice." _____ "and", "some", "mice" in "cats, dogs and some mice." |
| (?<! subexpression) | Zero-width negative lookbehind assertion. | \b\w+\b(?<! .+and.+) _____ \b\w+\b(?<! .+and.*) | "cats", "dogs", "and" in "cats, dogs and some mice." _____ "cats", "dogs" in "cats, dogs and some mice." |
| (?> subexpression) | Atomic group. | (?>a ab)c | "ac" in "ac" nothing in "abc" |

Lookarounds at a glance

When the regular expression engine hits a **lookaround expression**, it takes a substring reaching from the current position to the start (lookbehind) or end (lookahead) of the original string, and then runs [Regex.IsMatch](#) on that substring using the lookahead pattern. Success of this subexpression's result is then determined by whether it's a positive or negative assertion.

| Lookaround | Name | Function |
|------------|--------------------|---|
| (?=check) | Positive Lookahead | Asserts that what immediately follows the current position in the string is "check" |

| Lookaround | Name | Function |
|-------------|---------------------|--|
| (?<=>check) | Positive Lookbehind | Asserts that what immediately precedes the current position in the string is "check" |
| (?!=check) | Negative Lookahead | Asserts that what immediately follows the current position in the string is not "check" |
| (?<!check) | Negative Lookbehind | Asserts that what immediately precedes the current position in the string is not "check" |

Once they have matched, **atomic groups** won't be re-evaluated again, even when the remainder of the pattern fails due to the match. This can significantly improve performance when quantifiers occur within the atomic group or the remainder of the pattern.

Quantifiers

A quantifier specifies how many instances of the previous element (which can be a character, a group, or a character class) must be present in the input string for a match to occur. Quantifiers include the language elements listed in the following table. For more information, see [Quantifiers](#).

| Quantifier | Description | Pattern | Matches |
|------------|--|---------------------|--|
| * | Matches the previous element zero or more times. | a.*c | "abcabc" in "abcabc" |
| + | Matches the previous element one or more times. | "be+" | "bee" in "been", "be" in "bent" |
| ? | Matches the previous element zero or one time. | "rai?" | "rai" in "rain" |
| { n } | Matches the previous element exactly n times. | ",\d{3}" | ",043" in "1,043.6", ",876", ",543", and ",210" in "9,876,543,210" |
| { n , } | Matches the previous element at least n times. | "166", "29", "1930" | |
| { n , m } | Matches the previous element at least n times, but no more than m times. | \d{3,5} | "166", "17668", "19302" in "193024" |
| *? | Matches the previous element zero or more times, but as few | a.*?c | "abc" in "abcabc" |

| Quantifier | Description | Pattern | Matches |
|------------|---|---------------------|--|
| | times as possible. | | |
| +? | Matches the previous element one or more times, but as few times as possible. | "be+?" | "be" in "been", "be" in "bent" |
| ?? | Matches the previous element zero or one time, but as few times as possible. | "rai??" | "ra" in "rain" |
| { n }? | Matches the preceding element exactly <i>n</i> times. | ",\d{3}?" | ",043" in "1,043.6", ",876", ",543", and ",210" in "9,876,543,210" |
| { n , }? | Matches the previous element at least <i>n</i> times, but as few times as possible. | "166", "29", "1930" | |
| { n , m } | Matches the previous element between <i>n</i> and <i>m</i> times, but as few times as possible. | \d{3,5}?" | "166", "17668" |

Backreference Constructs

A backreference allows a previously matched subexpression to be identified subsequently in the same regular expression. The following table lists the backreference constructs supported by regular expressions in .NET. For more information, see [Backreference Constructs](#).

| Backreference construct | Description | Pattern | Matches |
|-------------------------|---|--------------------------------------|----------------|
| \ <i>number</i> | Backreference. Matches the value of a numbered subexpression. | (\w)\1 | "ee" in "seek" |
| \k< <i>name</i> > | Named backreference. Matches the value of a named expression. | (? <i>char</i>)\w)\k< <i>char</i> > | "ee" in "seek" |

Alternation Constructs

Alternation constructs modify a regular expression to enable either/or matching. These constructs include the language elements listed in the following table. For more information, see [Alternation Constructs](#).

| Alternation construct | Description | Pattern | Matches |
|---|--|---|---|
| <code> </code> | Matches any one element separated by the vertical bar (<code> </code>) character. | <code>th(e is at)</code> | "the", "this" in "this is the day." |
| <code>(?(<expression>) yes no)</code> or <code>(?(<expression>) yes)</code> | Matches <i>yes</i> if the regular expression pattern designated by <i>expression</i> matches; otherwise, matches the optional <i>no</i> part. <i>expression</i> is interpreted as a zero-width assertion. | <code>(? (A)A\d{2}\b \b\d{3}\b)</code> | "A10", "910" in "A10 C103 910" |
| | To avoid ambiguity with a named or numbered capturing group, you can optionally use an explicit assertion, like this: <code>(?(<= expression>) yes no)</code> | | |
| <code>(?(<name>) yes no)</code> or <code>(?(<name>) yes)</code> | Matches <i>yes</i> if <i>name</i> , a named or numbered capturing group, has a match; otherwise, matches the optional <i>no</i> . | <code>(?<quoted>)?(? (quoted).+?" \s+\s)</code> | "Dogs.jpg ", "\\"Yiska playing.jpg"" in "Dogs.jpg \"Yiska playing.jpg"" |

Substitutions

Substitutions are regular expression language elements that are supported in replacement patterns. For more information, see [Substitutions](#). The metacharacters listed in the following table are atomic zero-width assertions.

| Character | Description | Pattern | Replacement pattern | Input string | Result string |
|--------------------------|--|---|--|------------------------|---------------|
| <code>\$ number</code> | Substitutes the substring matched by group <i>number</i> . | <code>\b(\w+)(\s) (\w+)\b</code> | <code>\$3\$2\$1</code> | "one two" | "two one" |
| <code> \${ name }</code> | Substitutes the substring matched by the | <code>\b(?<word1>\w+) (\s)(? <word2>\w+)\b</code> | <code> \${word2} \${word1}</code> | "one two" "two one" | "two one" |

| Character | Description | Pattern | Replacement pattern | Input string | Result string |
|-----------|--|------------------|---------------------|--------------|---------------|
| | named group <i>name</i> . | | | | |
| \$\$ | Substitutes a literal "\$". | \b(\d+)\s?USD | \$\$\\$1 | "103 USD" | "\$103" |
| \$& | Substitutes a copy of the whole match. | \\$?\d*\.\.? \d+ | **\$&** | "\$1.30" | "**\$1.30**" |
| \$` | Substitutes all the text of the input string before the match. | B+ | \$` | "AABBCC" | "AAAACC" |
| \$' | Substitutes all the text of the input string after the match. | B+ | \$' | "AABBCC" | "AACCCC" |
| \$+ | Substitutes the last group that was captured. | B+(C+) | \$+ | "AABBCCDD" | "AACCDD" |
| \$_ | Substitutes the entire input string. | B+ | \$_ | "AABBCC" | "AAAABBCCCC" |

Regular Expression Options

You can specify options that control how the regular expression engine interprets a regular expression pattern. Many of these options can be specified either inline (in the regular expression pattern) or as one or more [RegexOptions](#) constants. This quick reference lists only inline options. For more information about inline and [RegexOptions](#) options, see the article [Regular Expression Options](#).

You can specify an inline option in two ways:

- By using the [miscellaneous construct](#) `(?imnsx-imnsx)`, where a minus sign (-) before an option or set of options turns those options off. For example, `(?i-mn)` turns case-insensitive matching (`i`) on, turns multiline mode (`m`) off, and turns unnamed group captures (`n`) off. The option applies to the regular expression pattern from the point at which the option is defined, and is effective either to the end of the pattern or to the point where another construct reverses the option.

- By using the [grouping construct](#) (`(?imnsx-imnsx: subexpression)`), which defines options for the specified group only.

The .NET regular expression engine supports the following inline options:

| Option | Description | Pattern | Matches |
|----------------|---|---------------------------------|--|
| <code>i</code> | Use case-insensitive matching. | <code>\b(?i)a(?-i)a\w+\b</code> | "aardvark", "aaaAuto" in "aardvark AAAuto aaaAuto Adam breakfast" |
| <code>m</code> | Use multiline mode. <code>^</code> and <code>\$</code> match the beginning and end of a line, instead of the beginning and end of a string. | | For an example, see the "Multiline Mode" section in Regular Expression Options . |
| <code>n</code> | Do not capture unnamed groups. | | For an example, see the "Explicit Captures Only" section in Regular Expression Options . |
| <code>s</code> | Use single-line mode. | | For an example, see the "Single-line Mode" section in Regular Expression Options . |
| <code>x</code> | Ignore unescaped white space in the regular expression pattern. | <code>\b(?x) \d+ \s \w+</code> | "1 aardvark", "2 cats" in "1 aardvark 2 cats IV centurions" |

Miscellaneous Constructs

Miscellaneous constructs either modify a regular expression pattern or provide information about it. The following table lists the miscellaneous constructs supported by .NET. For more information, see [Miscellaneous Constructs](#).

| Construct | Definition | Example |
|-----------------------------|--|--|
| <code>(?imnsx-imnsx)</code> | Sets or disables options such as case insensitivity in the middle of a pattern. For more information, see Regular Expression Options . | <code>\bA(?i)b\w+\b</code> matches "ABA", "Able" in "ABA Able Act" |
| <code>(?# comment)</code> | Inline comment. The comment ends at the first closing parenthesis. | <code>\bA(?#Matches words starting with A)\w+\b</code> |

| Construct | Definition | Example |
|--------------------|--|--|
| # [to end of line] | X-mode comment. The comment starts at an unescaped # and continues to the end of the line. | (?x)\bA\w+\b#Matches words starting with A |

See also

- [System.Text.RegularExpressions](#)
- [System.Text.RegularExpressions.Regex](#)
- [Regular Expressions](#)
- [Regular Expression Classes](#)
- [Regular Expressions - Quick Reference \(download in Word format\)](#) ↗
- [Regular Expressions - Quick Reference \(download in PDF format\)](#) ↗

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

Character Escapes in Regular Expressions

Article • 09/15/2021

The backslash (\) in a regular expression indicates one of the following:

- The character that follows it is a special character, as shown in the table in the following section. For example, `\b` is an anchor that indicates that a regular expression match should begin on a word boundary, `\t` represents a tab, and `\x020` represents a space.
- A character that otherwise would be interpreted as an unescaped language construct should be interpreted literally. For example, a brace (`{`) begins the definition of a quantifier, but a backslash followed by a brace (`\{`) indicates that the regular expression engine should match the brace. Similarly, a single backslash marks the beginning of an escaped language construct, but two backslashes (`\\"`) indicate that the regular expression engine should match the backslash.

ⓘ Note

Character escapes are recognized in regular expression patterns but not in replacement patterns.

Character Escapes in .NET

The following table lists the character escapes supported by regular expressions in .NET.

| Character or sequence | Description |
|--|--|
| All characters except for the following: <code>.</code> <code>\$</code> <code>^</code> <code>{</code> <code>[</code> <code>(</code> <code> </code> <code>)</code> <code>*</code> <code>+</code> <code>?</code> <code>\</code> | Characters other than those listed in the Character or sequence column have no special meaning in regular expressions; they match themselves. The characters included in the Character or sequence column are special regular expression language elements. To match them in a regular expression, they must be escaped or included in a positive character group . For example, the regular expression <code>\\$\d+</code> or <code>[\$]\d+</code> matches "\$1200". |
| <code>\a</code> | Matches a bell (alarm) character, <code>\u0007</code> . |

| Character or sequence | Description |
|-----------------------|---|
| \b | In a [character_group] character class, matches a backspace, \u0008. (See Character Classes .) Outside a character class, \b is an anchor that matches a word boundary. (See Anchors .) |
| \t | Matches a tab, \u0009. |
| \r | Matches a carriage return, \u000D. Note that \r is not equivalent to the newline character, \n. |
| \v | Matches a vertical tab, \u000B. |
| \f | Matches a form feed, \u000C. |
| \n | Matches a new line, \u000A. |
| \e | Matches an escape, \u001B. |
| \ nnn | Matches an ASCII character, where <i>nnn</i> consists of two or three digits that represent the octal character code. For example, \040 represents a space character. This construct is interpreted as a backreference if it has only one digit (for example, \2) or if it corresponds to the number of a capturing group. (See Backreference Constructs .) |
| \x nn | Matches an ASCII character, where <i>nn</i> is a two-digit hexadecimal character code. |
| \c X | Matches an ASCII control character, where <i>X</i> is the letter of the control character. For example, \cc is CTRL-C. |
| \u nnnn | Matches a UTF-16 code unit whose value is <i>nnnn</i> hexadecimal. Note: The Perl 5 character escape that is used to specify Unicode is not supported by .NET. The Perl 5 character escape has the form \x{ #####... }, where #####... is a series of hexadecimal digits. Instead, use \u <i>nnnn</i> . |
| \ | When followed by a character that is not recognized as an escaped character, matches that character. For example, * matches an asterisk (*) and is the same as \x2A. |

An Example

The following example illustrates the use of character escapes in a regular expression. It parses a string that contains the names of the world's largest cities and their populations in 2009. Each city name is separated from its population by a tab (\t) or a vertical bar (| or \u007c). Individual cities and their populations are separated from each other by a carriage return and line feed.

C#

```
using System;
using System.Text.RegularExpressions;

public class Example
{
    public static void Main()
    {
        string delimited = @"\G(.+)[\t\u007c](.+)\r?\n";
        string input = "Mumbai, India|13,922,125\t\n" +
                      "Shanghai, China\t13,831,900\n" +
                      "Karachi, Pakistan|12,991,000\n" +
                      "Delhi, India\t12,259,230\n" +
                      "Istanbul, Türkiye|11,372,613\n";
        Console.WriteLine("Population of the World's Largest Cities, 2009");
        Console.WriteLine();
        Console.WriteLine("{0,-20} {1,10}", "City", "Population");
        Console.WriteLine();
        foreach (Match match in Regex.Matches(input, delimited))
            Console.WriteLine("{0,-20} {1,10}", match.Groups[1].Value,
                             match.Groups[2].Value);
    }
}

// The example displays the following output:
//      Population of the World's Largest Cities, 2009
//
//      City          Population
//
//      Mumbai, India      13,922,125
//      Shanghai, China     13,831,900
//      Karachi, Pakistan   12,991,000
//      Delhi, India        12,259,230
//      Istanbul, Türkiye   11,372,613
```

The regular expression `\G(.+)[\t\u007c](.+)\r?\n` is interpreted as shown in the following table.

| Pattern | Description |
|-------------------------|--|
| <code>\G</code> | Begin the match where the last match ended. |
| <code>(.+)</code> | Match any character one or more times. This is the first capturing group. |
| <code>[\t\u007c]</code> | Match a tab (<code>\t</code>) or a vertical bar (<code> </code>). |
| <code>(.+)</code> | Match any character one or more times. This is the second capturing group. |
| <code>\r?\n</code> | Match zero or one occurrence of a carriage return followed by a new line. |

See also

- Regular Expression Language - Quick Reference

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

Character classes in regular expressions

Article • 06/27/2023

A character class defines a set of characters, any one of which can occur in an input string for a match to succeed. The regular expression language in .NET supports the following character classes:

- Positive character groups. A character in the input string must match one of a specified set of characters. For more information, see [Positive Character Group](#).
- Negative character groups. A character in the input string must not match one of a specified set of characters. For more information, see [Negative Character Group](#).
- Any character. The `.` (dot or period) character in a regular expression is a wildcard character that matches any character except `\n`. For more information, see [Any Character](#).
- A general Unicode category or named block. A character in the input string must be a member of a particular Unicode category or must fall within a contiguous range of Unicode characters for a match to succeed. For more information, see [Unicode Category or Unicode Block](#).
- A negative general Unicode category or named block. A character in the input string must not be a member of a particular Unicode category or must not fall within a contiguous range of Unicode characters for a match to succeed. For more information, see [Negative Unicode Category or Unicode Block](#).
- A word character. A character in the input string can belong to any of the Unicode categories that are appropriate for characters in words. For more information, see [Word Character](#).
- A non-word character. A character in the input string can belong to any Unicode category that is not a word character. For more information, see [Non-Word Character](#).
- A white-space character. A character in the input string can be any Unicode separator character, as well as any one of a number of control characters. For more information, see [White-Space Character](#).
- A non-white-space character. A character in the input string can be any character that is not a white-space character. For more information, see [Non-White-Space Character](#).

- A decimal digit. A character in the input string can be any of a number of characters classified as Unicode decimal digits. For more information, see [Decimal Digit Character](#).
- A non-decimal digit. A character in the input string can be anything other than a Unicode decimal digit. For more information, see [Decimal Digit Character](#).

.NET supports character class subtraction expressions, which enables you to define a set of characters as the result of excluding one character class from another character class. For more information, see [Character Class Subtraction](#).

 **Note**

Character classes that match characters by category, such as `\w` to match word characters or `\p{}` to match a Unicode category, rely on the `CharUnicodeInfo` class to provide information about character categories. In .NET Framework 4.6.2 and later versions, character categories are based on [The Unicode Standard, Version 8.0.0](#).

Positive character group: []

A positive character group specifies a list of characters, any one of which may appear in an input string for a match to occur. This list of characters may be specified individually, as a range, or both.

The syntax for specifying a list of individual characters is as follows:

```
[*character_group*]
```

where *character_group* is a list of the individual characters that can appear in the input string for a match to succeed. *character_group* can consist of any combination of one or more literal characters, [escape characters](#), or character classes.

The syntax for specifying a range of characters is as follows:

```
[firstCharacter-lastCharacter]
```

where *firstCharacter* is the character that begins the range and *lastCharacter* is the character that ends the range. A character range is a contiguous series of characters defined by specifying the first character in the series, a hyphen (-), and then the last character in the series. Two characters are contiguous if they have adjacent Unicode

code points. *firstCharacter* must be the character with the lower code point, and *lastCharacter* must be the character with the higher code point.

ⓘ Note

Because a positive character group can include both a set of characters and a character range, a hyphen character (-) is always interpreted as the range separator unless it is the first or last character of the group.

To include a hyphen as a nonperipheral member of a character group, escape it. For instance, to create a character group for the character a and the characters from - to /, the correct syntax is [a\--/].

Some common regular expression patterns that contain positive character classes are listed in the following table.

| Pattern | Description |
|-----------|---|
| [aeiou] | Match all vowels. |
| [\p{P}\d] | Match all punctuation and decimal digit characters. |
| [\s\p{P}] | Match all white space and punctuation. |

The following example defines a positive character group that contains the characters "a" and "e" so that the input string must contain the words "grey" or "gray" followed by another word for a match to occur.

C#

```
using System;
using System.Text.RegularExpressions;

public class Example
{
    public static void Main()
    {
        string pattern = @"gr[ae]y\s\S+?[\s\p{P}]";
        string input = "The gray wolf jumped over the grey wall.";
        MatchCollection matches = Regex.Matches(input, pattern);
        foreach (Match match in matches)
            Console.WriteLine($"'{match.Value}'");
    }
}
// The example displays the following output:
```

```
//      'gray wolf '
//      'grey wall.'
```

The regular expression `gr[ae]y\s\S+?[\s|\p{P}]` is defined as follows:

| Pattern | Description |
|------------------------|---|
| <code>gr</code> | Match the literal characters "gr". |
| <code>[ae]</code> | Match either an "a" or an "e". |
| <code>y\s</code> | Match the literal character "y" followed by a white-space character. |
| <code>\S+?</code> | Match one or more non-white-space characters, but as few as possible. |
| <code>[\s\p{P}]</code> | Match either a white-space character or a punctuation mark. |

The following example matches words that begin with any capital letter. It uses the subexpression `[A-Z]` to represent the range of capital letters from A to Z.

C#

```
using System;
using System.Text.RegularExpressions;

public class Example
{
    public static void Main()
    {
        string pattern = @"\b[A-Z]\w*\b";
        string input = "A city Albany Zulu maritime Marseilles";
        foreach (Match match in Regex.Matches(input, pattern))
            Console.WriteLine(match.Value);
    }
}
// The example displays the following output:
//      A
//      Albany
//      Zulu
//      Marseilles
```

The regular expression `\b[A-Z]\w*\b` is defined as shown in the following table.

| Pattern | Description |
|--------------------|--|
| <code>\b</code> | Start at a word boundary. |
| <code>[A-Z]</code> | Match any uppercase character from A to Z. |
| <code>\w*</code> | Match zero or more word characters. |

| Pattern | Description |
|---------|------------------------|
| \b | Match a word boundary. |

Negative character group: [^]

A negative character group specifies a list of characters that must not appear in an input string for a match to occur. The list of characters may be specified individually, as a range, or both.

The syntax for specifying a list of individual characters is as follows:

```
[*^character_group*]
```

where *character_group* is a list of the individual characters that cannot appear in the input string for a match to succeed. *character_group* can consist of any combination of one or more literal characters, [escape characters](#), or character classes.

The syntax for specifying a range of characters is as follows:

```
[^*firstCharacter*-*lastCharacter*]
```

where *firstCharacter* is the character that begins the range and *lastCharacter* is the character that ends the range. A character range is a contiguous series of characters defined by specifying the first character in the series, a hyphen (-), and then the last character in the series. Two characters are contiguous if they have adjacent Unicode code points. *firstCharacter* must be the character with the lower code point, and *lastCharacter* must be the character with the higher code point.

Note

Because a negative character group can include both a set of characters and a character range, a hyphen character (-) is always interpreted as the range separator unless it is the first or last character of the group.

Two or more character ranges can be concatenated. For example, to specify the range of decimal digits from "0" through "9", the range of lowercase letters from "a" through "f", and the range of uppercase letters from "A" through "F", use `[0-9a-fA-F]`.

The leading caret character (^) in a negative character group is mandatory and indicates the character group is a negative character group instead of a positive character group.

Important

A negative character group in a larger regular expression pattern is not a zero-width assertion. That is, after evaluating the negative character group, the regular expression engine advances one character in the input string.

Some common regular expression patterns that contain negative character groups are listed in the following table.

| Pattern | Description |
|------------|---|
| [^aeiou] | Match all characters except vowels. |
| [^\p{P}\d] | Match all characters except punctuation and decimal digit characters. |

The following example matches any word that begins with the characters "th" and is not followed by an "o".

C#

```
using System;
using System.Text.RegularExpressions;

public class Example
{
    public static void Main()
    {
        string pattern = @"\bth[^o]\w+\b";
        string input = "thought thing though them through thus thorough this";
        foreach (Match match in Regex.Matches(input, pattern))
            Console.WriteLine(match.Value);
    }
}
// The example displays the following output:
//      thing
//      them
//      through
//      thus
//      this
```

The regular expression `\bth[^o]\w+\b` is defined as shown in the following table.

| Pattern | Description |
|---------|---|
| \b | Start at a word boundary. |
| th | Match the literal characters "th". |
| [^o] | Match any character that is not an "o". |

| Pattern | Description |
|---------|------------------------------------|
| \w+ | Match one or more word characters. |
| \b | End at a word boundary. |

Any character: .

The period character (.) matches any character except `\n` (the newline character), with the following two qualifications:

- If a regular expression pattern is modified by the [RegexOptions.Singleline](#) option, or if the portion of the pattern that contains the `.` character class is modified by the `s` option, `.` matches any character. For more information, see [Regular Expression Options](#).

The following example illustrates the different behavior of the `.` character class by default and with the [RegexOptions.Singleline](#) option. The regular expression `^.+` starts at the beginning of the string and matches every character. By default, the match ends at the end of the first line; the regular expression pattern matches the carriage return character, `\r`, but it does not match `\n`. Because the [RegexOptions.Singleline](#) option interprets the entire input string as a single line, it matches every character in the input string, including `\n`.

```
C#
using System;
using System.Text.RegularExpressions;

public class Example
{
    public static void Main()
    {
        string pattern = "^.+";
        string input = "This is one line and" + Environment.NewLine +
"this is the second.";
        foreach (Match match in Regex.Matches(input, pattern))
            Console.WriteLine(Regex.Escape(match.Value));

        Console.WriteLine();
        foreach (Match match in Regex.Matches(input, pattern,
RegexOptions.Singleline))
            Console.WriteLine(Regex.Escape(match.Value));
    }
}
// The example displays the following output:
//      This\ is\ one\ line\ and\r
```

```
//  
//      This\ is\ one\ line\ and\r\nthis\ is\ the\ second\.
```

ⓘ Note

Because it matches any character except `\n`, the `.` character class also matches `\r` (the carriage return character).

- In a positive or negative character group, a period is treated as a literal period character, and not as a character class. For more information, see [Positive Character Group](#) and [Negative Character Group](#) earlier in this topic. The following example provides an illustration by defining a regular expression that includes the period character `(.)` both as a character class and as a member of a positive character group. The regular expression `\b.*[.?!;:](\s|\z)` begins at a word boundary, matches any character until it encounters one of five punctuation marks, including a period, and then matches either a white-space character or the end of the string.

```
C#  
  
using System;  
using System.Text.RegularExpressions;  
  
public class Example  
{  
    public static void Main()  
    {  
        string pattern = @"\b.*[.?!;:](\s|\z)";  
        string input = "this. what: is? go, thing.";  
        foreach (Match match in Regex.Matches(input, pattern))  
            Console.WriteLine(match.Value);  
    }  
}  
// The example displays the following output:  
//      this. what: is? go, thing.
```

ⓘ Note

Because it matches any character, the `.` language element is often used with a lazy quantifier if a regular expression pattern attempts to match any character multiple times. For more information, see [Quantifiers](#).

Unicode category or Unicode block: \p{}

The Unicode standard assigns each character a general category. For example, a particular character can be an uppercase letter (represented by the `Lu` category), a decimal digit (the `Nd` category), a math symbol (the `Sm` category), or a paragraph separator (the `Z1` category). Specific character sets in the Unicode standard also occupy a specific range or block of consecutive code points. For example, the basic Latin character set is found from `\u0000` through `\u007F`, while the Arabic character set is found from `\u0600` through `\u06FF`.

The regular expression construct

```
\p{ name }
```

matches any character that belongs to a Unicode general category or named block, where *name* is the category abbreviation or named block name. For a list of category abbreviations, see the [Supported Unicode General Categories](#) section later in this topic. For a list of named blocks, see the [Supported Named Blocks](#) section later in this topic.

Tip

Matching may be improved if the string is first normalized by calling the `String.Normalize` method.

The following example uses the `\p{ name }` construct to match both a Unicode general category (in this case, the `Pd`, or Punctuation, Dash category) and a named block (the `IsGreek` and `IsBasicLatin` named blocks).

C#

```
using System;
using System.Text.RegularExpressions;

public class Example
{
    public static void Main()
    {
        string pattern = @"\b(\p{IsGreek}+(\s)?)+\p{Pd}\s(\p{IsBasicLatin}+
(\s)?)+";
        string input = "Κατα Μαθηταιον - The Gospel of Matthew";

        Console.WriteLine(Regex.IsMatch(input, pattern));           // Displays
        True.
```

```
}
```

The regular expression `\b(\p{IsGreek}+(\s)?)+\p{Pd}\s(\p{IsBasicLatin}+(\s)?)+` is defined as shown in the following table.

| Pattern | Description |
|--|---|
| <code>\b</code> | Start at a word boundary. |
| <code>\p{IsGreek}+</code> | Match one or more Greek characters. |
| <code>(\s)?</code> | Match zero or one white-space character. |
| <code>(\p{IsGreek}+(\s)?)+</code> | Match the pattern of one or more Greek characters followed by zero or one white-space characters one or more times. |
| <code>\p{Pd}</code> | Match a Punctuation, Dash character. |
| <code>\s</code> | Match a white-space character. |
| <code>\p{IsBasicLatin}+</code> | Match one or more basic Latin characters. |
| <code>(\s)?</code> | Match zero or one white-space character. |
| <code>(\p{IsBasicLatin}+(\s)?)+</code> | Match the pattern of one or more basic Latin characters followed by zero or one white-space characters one or more times. |

Negative Unicode category or Unicode block: `\P{}`

The Unicode standard assigns each character a general category. For example, a particular character can be an uppercase letter (represented by the `Lu` category), a decimal digit (the `Nd` category), a math symbol (the `Sm` category), or a paragraph separator (the `Zl` category). Specific character sets in the Unicode standard also occupy a specific range or block of consecutive code points. For example, the basic Latin character set is found from `\u0000` through `\u007F`, while the Arabic character set is found from `\u0600` through `\u06FF`.

The regular expression construct

```
\P{ name }
```

matches any character that does not belong to a Unicode general category or named block, where *name* is the category abbreviation or named block name. For a list of

category abbreviations, see the [Supported Unicode General Categories](#) section later in this topic. For a list of named blocks, see the [Supported Named Blocks](#) section later in this topic.

Tip

Matching may be improved if the string is first normalized by calling the [String.Normalize](#) method.

The following example uses the `\P{ name }` construct to remove any currency symbols (in this case, the `Sc`, or Symbol, Currency category) from numeric strings.

C#

```
using System;
using System.Text.RegularExpressions;

public class Example
{
    public static void Main()
    {
        string pattern = @"\P{Sc}+";

        string[] values = { "$164,091.78", "£1,073,142.68", "73¢", "€120" };
        foreach (string value in values)
            Console.WriteLine(Regex.Match(value, pattern).Value);
    }
}
// The example displays the following output:
//      164,091.78
//      1,073,142.68
//      73
//      120
```

The regular expression pattern `(\P{Sc})+` matches one or more characters that are not currency symbols; it effectively strips any currency symbol from the result string.

Word character: \w

`\w` matches any word character. A word character is a member of any of the Unicode categories listed in the following table.

| Category | Description |
|----------|-------------------|
| L | Letter, Lowercase |

| Category | Description |
|----------|--|
| Lu | Letter, Uppercase |
| Lt | Letter, Titlecase |
| Lo | Letter, Other |
| Lm | Letter, Modifier |
| Mn | Mark, Nonspacing |
| Nd | Number, Decimal Digit |
| Pc | Punctuation, Connector. This category includes ten characters, the most commonly used of which is the LOWLINE character (_), u+005F. |

If ECMAScript-compliant behavior is specified, `\w` is equivalent to `[a-zA-Z_0-9]`. For information on ECMAScript regular expressions, see the "ECMAScript Matching Behavior" section in [Regular Expression Options](#).

ⓘ Note

Because it matches any word character, the `\w` language element is often used with a lazy quantifier if a regular expression pattern attempts to match any word character multiple times, followed by a specific word character. For more information, see [Quantifiers](#).

The following example uses the `\w` language element to match duplicate characters in a word. The example defines a regular expression pattern, `(\w)\1`, which can be interpreted as follows.

| Element | Description |
|-------------------|--|
| <code>(\w)</code> | Match a word character. This is the first capturing group. |
| <code>\1</code> | Match the value of the first capture. |

C#

```
using System;
using System.Text.RegularExpressions;

public class Example
{
    public static void Main()
    {
```

```

        string pattern = @"(\w)\1";
        string[] words = { "trellis", "seer", "latter", "summer",
                           "hoarse", "lesser", "aardvark", "stunned" };
        foreach (string word in words)
        {
            Match match = Regex.Match(word, pattern);
            if (match.Success)
                Console.WriteLine($"'{0}' found in '{1}' at position {2}.",
                                  match.Value, word, match.Index);
            else
                Console.WriteLine("No double characters in '{0}'.", word);
        }
    }
}

// The example displays the following output:
//      'll' found in 'trellis' at position 3.
//      'ee' found in 'seer' at position 1.
//      'tt' found in 'latter' at position 2.
//      'mm' found in 'summer' at position 2.
//      No double characters in 'hoarse'.
//      'ss' found in 'lesser' at position 2.
//      'aa' found in 'aardvark' at position 0.
//      'nn' found in 'stunned' at position 3.

```

Non-word character: \W

\W matches any non-word character. The \W language element is equivalent to the following character class:

```
[^\p{Ll}\p{Lu}\p{Lt}\p{Lo}\p{Nd}\p{Pc}\p{Lm}]
```

In other words, it matches any character except for those in the Unicode categories listed in the following table.

| Category | Description |
|----------|-----------------------|
| Ll | Letter, Lowercase |
| Lu | Letter, Uppercase |
| Lt | Letter, Titlecase |
| Lo | Letter, Other |
| Lm | Letter, Modifier |
| Mn | Mark, Nonspacing |
| Nd | Number, Decimal Digit |

| Category | Description |
|----------|--|
| Pc | Punctuation, Connector. This category includes ten characters, the most commonly used of which is the LOWLINE character (_), u+005F. |

If ECMAScript-compliant behavior is specified, `\w` is equivalent to `[^a-zA-Z_0-9]`. For information on ECMAScript regular expressions, see the "ECMAScript Matching Behavior" section in [Regular Expression Options](#).

ⓘ Note

Because it matches any non-word character, the `\w` language element is often used with a lazy quantifier if a regular expression pattern attempts to match any non-word character multiple times followed by a specific non-word character. For more information, see [Quantifiers](#).

The following example illustrates the `\w` character class. It defines a regular expression pattern, `\b(\w+)(\w){1,2}`, that matches a word followed by one or two non-word characters, such as white space or punctuation. The regular expression is interpreted as shown in the following table.

| Element | Description |
|------------------------|---|
| <code>\b</code> | Begin the match at a word boundary. |
| <code>(\w+)</code> | Match one or more word characters. This is the first capturing group. |
| <code>(\w){1,2}</code> | Match a non-word character either one or two times. This is the second capturing group. |

C#

```
using System;
using System.Text.RegularExpressions;

public class Example
{
    public static void Main()
    {
        string pattern = @"\b(\w+)(\w){1,2}";
        string input = "The old, grey mare slowly walked across the narrow,
green pasture.";
        foreach (Match match in Regex.Matches(input, pattern))
        {
            Console.WriteLine(match.Value);
            Console.Write(" Non-word character(s):");
        }
    }
}
```

```

        CaptureCollection captures = match.Groups[2].Captures;
        for (int ctr = 0; ctr < captures.Count; ctr++)
            Console.WriteLine(@"'{0}' (\u{1}){2}", captures[ctr].Value,
Convert.ToInt16(captures[ctr].Value[0]).ToString("X4"),
                ctr < captures.Count - 1 ? ", " : "");
            Console.WriteLine();
        }
    }
// The example displays the following output:
//      The
//          Non-word character(s):' ' (\u0020)
//      old,
//          Non-word character(s):',' (\u002C), ' ' (\u0020)
//      grey
//          Non-word character(s):' ' (\u0020)
//      mare
//          Non-word character(s):' ' (\u0020)
//      slowly
//          Non-word character(s):' ' (\u0020)
//      walked
//          Non-word character(s):' ' (\u0020)
//      across
//          Non-word character(s):' ' (\u0020)
//      the
//          Non-word character(s):' ' (\u0020)
//      narrow,
//          Non-word character(s):',' (\u002C), ' ' (\u0020)
//      green
//          Non-word character(s):' ' (\u0020)
//      pasture.
//          Non-word character(s):'.' (\u002E)

```

Because the [Group](#) object for the second capturing group contains only a single captured non-word character, the example retrieves all captured non-word characters from the [CaptureCollection](#) object that is returned by the [Group.Captures](#) property.

Whitespace character: \s

\s matches any whitespace character. It is equivalent to the escape sequences and Unicode categories listed in the following table.

| Category | Description |
|----------|--|
| \f | The form feed character, \u000C. |
| \n | The newline character, \u000A. |
| \r | The carriage return character, \u000D. |

| Category | Description |
|----------|--|
| \t | The tab character, \u0009. |
| \v | The vertical tab character, \u000B. |
| \x85 | The NEXT LINE (NEL) character, \u0085. |
| \p{Z} | Matches all separator characters . This includes the <code>zs</code> , <code>z1</code> , and <code>Zp</code> categories. |

If ECMAScript-compliant behavior is specified, `\s` is equivalent to `[\f\n\r\t\v]`. For information on ECMAScript regular expressions, see the "ECMAScript Matching Behavior" section in [Regular Expression Options](#).

The following example illustrates the `\s` character class. It defines a regular expression pattern, `\b\w+(e)?s(\s|$)`, that matches a word ending in either "s" or "es" followed by either a white-space character or the end of the input string. The regular expression is interpreted as shown in the following table.

| Element | Description |
|---------|--|
| \b | Begin the match at a word boundary. |
| \w+ | Match one or more word characters. |
| (e)? | Match an "e" either zero or one time. |
| s | Match an "s". |
| (\s \$) | Match either a white-space character or the end of the input string. |

C#

```

using System;
using System.Text.RegularExpressions;

public class Example
{
    public static void Main()
    {
        string pattern = @"\b\w+(e)?s(\s|$)";
        string input = "matches stores stops leave leaves";
        foreach (Match match in Regex.Matches(input, pattern))
            Console.WriteLine(match.Value);
    }
}
// The example displays the following output:
//      matches
//      stores

```

```
//      stops  
//      leaves
```

Non-whitespace character: \S

\s matches any non-white-space character. It is equivalent to the [^\f\n\r\t\v\x85\p{Z}] regular expression pattern, or the opposite of the regular expression pattern that is equivalent to \s, which matches white-space characters. For more information, see [White-Space Character: \s](#).

If ECMAScript-compliant behavior is specified, \s is equivalent to [^\f\n\r\t\v]. For information on ECMAScript regular expressions, see the "ECMAScript Matching Behavior" section in [Regular Expression Options](#).

The following example illustrates the \s language element. The regular expression pattern \b(\S+)\s? matches strings that are delimited by white-space characters. The second element in the match's [GroupCollection](#) object contains the matched string. The regular expression can be interpreted as shown in the following table.

| Element | Description |
|---------|--|
| \b | Begin the match at a word boundary. |
| (\S+) | Match one or more non-white-space characters. This is the first capturing group. |
| \s? | Match zero or one white-space character. |

C#

```
using System;  
using System.Text.RegularExpressions;  
  
public class Example  
{  
    public static void Main()  
    {  
        string pattern = @"\b(\S+)\s?";  
        string input = "This is the first sentence of the first paragraph. " +  
                      "This is the second sentence.\n" +  
                      "This is the only sentence of the second  
paragraph.";  
        foreach (Match match in Regex.Matches(input, pattern))  
            Console.WriteLine(match.Groups[1]);  
    }  
}  
// The example displays the following output:  
//      This
```

```
// is
// the
// first
// sentence
// of
// the
// first
// paragraph.
// This
// is
// the
// second
// sentence.
// This
// is
// the
// only
// sentence
// of
// the
// second
// paragraph.
```

Decimal digit character: \d

\d matches any decimal digit. It is equivalent to the \p{Nd} regular expression pattern, which includes the standard decimal digits 0-9 as well as the decimal digits of a number of other character sets.

If ECMAScript-compliant behavior is specified, \d is equivalent to [0-9]. For information on ECMAScript regular expressions, see the "ECMAScript Matching Behavior" section in [Regular Expression Options](#).

The following example illustrates the \d language element. It tests whether an input string represents a valid telephone number in the United States and Canada. The regular expression pattern ^(\(?(\d{3})\)?[\s-])?\d{3}-\d{4}\$ is defined as shown in the following table.

| Element | Description |
|---------|---|
| ^ | Begin the match at the beginning of the input string. |
| \(?) | Match zero or one literal "(" character. |
| \d{3} | Match three decimal digits. |
| \)?) | Match zero or one literal ")" character. |

| Element | Description |
|-------------------------|--|
| [\s-] | Match a hyphen or a white-space character. |
| (\(\?\d{3}\)\)?[\s-]?)? | Match an optional opening parenthesis followed by three decimal digits, an optional closing parenthesis, and either a white-space character or a hyphen zero or one time. This is the first capturing group. |
| \d{3}-\d{4} | Match three decimal digits followed by a hyphen and four more decimal digits. |
| \$ | Match the end of the input string. |

C#

```

using System;
using System.Text.RegularExpressions;

public class Example
{
    public static void Main()
    {
        string pattern = @"^(\(\?\d{3}\)\)?[\s-]?)?\d{3}-\d{4}$";
        string[] inputs = { "111 111-1111", "222-2222", "222 333-444",
                            "(212) 111-1111", "111-AB1-1111",
                            "212-111-1111", "01 999-9999" };

        foreach (string input in inputs)
        {
            if (Regex.IsMatch(input, pattern))
                Console.WriteLine(input + ": matched");
            else
                Console.WriteLine(input + ": match failed");
        }
    }
}

// The example displays the following output:
//      111 111-1111: matched
//      222-2222: matched
//      222 333-444: match failed
//      (212) 111-1111: matched
//      111-AB1-1111: match failed
//      212-111-1111: matched
//      01 999-9999: match failed

```

Non-digit character: \D

\D matches any non-digit character. It is equivalent to the \P{Nd} regular expression pattern.

If ECMAScript-compliant behavior is specified, `\D` is equivalent to `[^0-9]`. For information on ECMAScript regular expressions, see the "ECMAScript Matching Behavior" section in [Regular Expression Options](#).

The following example illustrates the `\D` language element. It tests whether a string such as a part number consists of the appropriate combination of decimal and non-decimal characters. The regular expression pattern `^\D\d{1,5}\D*$` is defined as shown in the following table.

| Element | Description |
|----------------------|---|
| <code>^</code> | Begin the match at the beginning of the input string. |
| <code>\D</code> | Match a non-digit character. |
| <code>\d{1,5}</code> | Match from one to five decimal digits. |
| <code>\D*</code> | Match zero, one, or more non-decimal characters. |
| <code>\$</code> | Match the end of the input string. |

C#

```
using System;
using System.Text.RegularExpressions;

public class Example
{
    public static void Main()
    {
        string pattern = @"^\D\d{1,5}\D*$";
        string[] inputs = { "A1039C", "AA0001", "C18A", "Y938518" };

        foreach (string input in inputs)
        {
            if (Regex.IsMatch(input, pattern))
                Console.WriteLine(input + ": matched");
            else
                Console.WriteLine(input + ": match failed");
        }
    }
}

// The example displays the following output:
//      A1039C: matched
//      AA0001: match failed
//      C18A: matched
//      Y938518: match failed
```

Supported Unicode general categories

Unicode defines the general categories listed in the following table. For more information, see the "UCD File Format" and "General Category Values" subtopics at the [Unicode Character Database](#), Sec. 5.7.1, Table 12.

| Category | Description |
|----------|---|
| Lu | Letter, Uppercase |
| Ll | Letter, Lowercase |
| Lt | Letter, Titlecase |
| Lm | Letter, Modifier |
| Lo | Letter, Other |
| L | All letter characters. This includes the Lu, Ll, Lt, Lm, and Lo characters. |
| Mn | Mark, Nonspacing |
| Mc | Mark, Spacing Combining |
| Me | Mark, Enclosing |
| M | All combining marks. This includes the Mn, Mc, and Me categories. |
| Nd | Number, Decimal Digit |
| Nl | Number, Letter |
| No | Number, Other |
| N | All numbers. This includes the Nd, Nl, and No categories. |
| Pc | Punctuation, Connector |
| Pd | Punctuation, Dash |
| Ps | Punctuation, Open |
| Pe | Punctuation, Close |
| Pi | Punctuation, Initial quote (may behave like Ps or Pe depending on usage) |
| Pf | Punctuation, Final quote (may behave like Ps or Pe depending on usage) |
| Po | Punctuation, Other |

| Category | Description |
|----------|---|
| P | All punctuation characters. This includes the <code>Pc</code> , <code>Pd</code> , <code>Ps</code> , <code>Pe</code> , <code>Pi</code> , <code>Pf</code> , and <code>Po</code> categories. |
| Sm | Symbol, Math |
| Sc | Symbol, Currency |
| Sk | Symbol, Modifier |
| So | Symbol, Other |
| S | All symbols. This includes the <code>Sm</code> , <code>Sc</code> , <code>Sk</code> , and <code>So</code> categories. |
| Zs | Separator, Space |
| Zl | Separator, Line |
| Zp | Separator, Paragraph |
| Z | All separator characters. This includes the <code>Zs</code> , <code>Zl</code> , and <code>Zp</code> categories. |
| Cc | Other, Control |
| Cf | Other, Format |
| Cs | Other, Surrogate |
| Co | Other, Private Use |
| Cn | Other, Not Assigned or Noncharacter |
| C | All other characters. This includes the <code>Cc</code> , <code>Cf</code> , <code>Cs</code> , <code>Co</code> , and <code>Cn</code> categories. |

You can determine the Unicode category of any particular character by passing that character to the [GetUnicodeCategory](#) method. The following example uses the [GetUnicodeCategory](#) method to determine the category of each element in an array that contains selected Latin characters.

C#

```
using System;
using System.Text.RegularExpressions;

public class Example
{
    public static void Main()
    {
        char[] chars = { 'a', 'X', '8', ',', ' ', '\u0009', '!' };
```

```

        foreach (char ch in chars)
            Console.WriteLine("{0}': {1}",
                Regex.Escape(ch.ToString()),
                Char.GetUnicodeCategory(ch));
    }
}

// The example displays the following output:
//      'a': LowercaseLetter
//      'X': UppercaseLetter
//      '8': DecimalDigitNumber
//      ',': OtherPunctuation
//      '\ ': SpaceSeparator
//      '\t': Control
//      '!': OtherPunctuation

```

Supported named blocks

.NET provides the named blocks listed in the following table. The set of supported named blocks is based on Unicode 4.0 and Perl 5.6. For a regular expression that uses named blocks, see the [Unicode category or Unicode block: \p{}](#) section.

| Code point range | Block name |
|------------------|-----------------------------|
| 0000 - 007F | IsBasicLatin |
| 0080 - 00FF | IsLatin-1Supplement |
| 0100 - 017F | IsLatinExtended-A |
| 0180 - 024F | IsLatinExtended-B |
| 0250 - 02AF | IsIPAExtensions |
| 02B0 - 02FF | IsSpacingModifierLetters |
| 0300 - 036F | IsCombiningDiacriticalMarks |
| 0370 - 03FF | IsGreek |
| -or- | |
| | IsGreekandCoptic |
| 0400 - 04FF | IsCyrillic |
| 0500 - 052F | IsCyrillicSupplement |
| 0530 - 058F | IsArmenian |
| 0590 - 05FF | IsHebrew |

| Code point range | Block name |
|-------------------------|--------------------------------------|
| 0600 - 06FF | IsArabic |
| 0700 - 074F | IsSyriac |
| 0780 - 07BF | IsThaana |
| 0900 - 097F | IsDevanagari |
| 0980 - 09FF | IsBengali |
| 0A00 - 0A7F | IsGurmukhi |
| 0A80 - 0AFF | IsGujarati |
| 0B00 - 0B7F | IsOriya |
| 0B80 - 0BFF | IsTamil |
| 0C00 - 0C7F | IsTelugu |
| 0C80 - 0CFF | IsKannada |
| 0D00 - 0D7F | IsMalayalam |
| 0D80 - 0DFF | IsSinhala |
| 0E00 - 0E7F | IsThai |
| 0E80 - 0EFF | IsLao |
| 0F00 - 0FFF | IsTibetan |
| 1000 - 109F | IsMyanmar |
| 10A0 - 10FF | IsGeorgian |
| 1100 - 11FF | IsHangulJamo |
| 1200 - 137F | IsEthiopic |
| 13A0 - 13FF | IsCherokee |
| 1400 - 167F | IsUnifiedCanadianAboriginalSyllabics |
| 1680 - 169F | IsOgham |
| 16A0 - 16FF | IsRunic |
| 1700 - 171F | IsTagalog |
| 1720 - 173F | IsHanunoo |

| Code point range | Block name |
|-------------------------|---------------------------------------|
| 1740 - 175F | IsBuhid |
| 1760 - 177F | IsTagbanwa |
| 1780 - 17FF | IsKhmer |
| 1800 - 18AF | IsMongolian |
| 1900 - 194F | IsLimbu |
| 1950 - 197F | IsTaiLe |
| 19E0 - 19FF | IsKhmerSymbols |
| 1D00 - 1D7F | IsPhoneticExtensions |
| 1E00 - 1EFF | IsLatinExtendedAdditional |
| 1F00 - 1FFF | IsGreekExtended |
| 2000 - 206F | IsGeneralPunctuation |
| 2070 - 209F | IsSuperscriptsandSubscripts |
| 20A0 - 20CF | IsCurrencySymbols |
| 20D0 - 20FF | IsCombiningDiacriticalMarksforSymbols |
| -or- | |
| | IsCombiningMarksforSymbols |
| 2100 - 214F | IsLetterlikeSymbols |
| 2150 - 218F | IsNumberForms |
| 2190 - 21FF | IsArrows |
| 2200 - 22FF | IsMathematicalOperators |
| 2300 - 23FF | IsMiscellaneousTechnical |
| 2400 - 243F | IsControlPictures |
| 2440 - 245F | IsOpticalCharacterRecognition |
| 2460 - 24FF | IsEnclosedAlphanumerics |
| 2500 - 257F | IsBoxDrawing |
| 2580 - 259F | IsBlockElements |

| | |
|-------------------------|--------------------------------------|
| Code point range | Block name |
| 25A0 - 25FF | IsGeometricShapes |
| 2600 - 26FF | IsMiscellaneousSymbols |
| 2700 - 27BF | IsDingbats |
| 27C0 - 27EF | IsMiscellaneousMathematicalSymbols-A |
| 27F0 - 27FF | IsSupplementalArrows-A |
| 2800 - 28FF | IsBraillePatterns |
| 2900 - 297F | IsSupplementalArrows-B |
| 2980 - 29FF | IsMiscellaneousMathematicalSymbols-B |
| 2A00 - 2AFF | IsSupplementalMathematicalOperators |
| 2B00 - 2BFF | IsMiscellaneousSymbolsandArrows |
| 2E80 - 2EFF | IsCJKRadicalsSupplement |
| 2F00 - 2FDF | IsKangxiRadicals |
| 2FF0 - 2FFF | IsIdeographicDescriptionCharacters |
| 3000 - 303F | IsCJKSymbolsandPunctuation |
| 3040 - 309F | IsHiragana |
| 30A0 - 30FF | IsKatakana |
| 3100 - 312F | IsBopomofo |
| 3130 - 318F | IsHangulCompatibilityJamo |
| 3190 - 319F | IsKanbun |
| 31A0 - 31BF | IsBopomofoExtended |
| 31F0 - 31FF | IsKatakanaPhoneticExtensions |
| 3200 - 32FF | IsEnclosedCJKLettersandMonths |
| 3300 - 33FF | IsCJKCompatibility |
| 3400 - 4DBF | IsCJKUnifiedIdeographsExtensionA |
| 4DC0 - 4DFF | IsYijingHexagramSymbols |
| 4E00 - 9FFF | IsCJKUnifiedIdeographs |

| Code point range | Block name |
|------------------|----------------------------------|
| A000 - A48F | IsYiSyllables |
| A490 - A4CF | IsYiRadicals |
| AC00 - D7AF | IsHangulSyllables |
| D800 - DB7F | IsHighSurrogates |
| DB80 - DBFF | IsHighPrivateUseSurrogates |
| DC00 - DFFF | IsLowSurrogates |
| E000 - F8FF | IsPrivateUse OR IsPrivateUseArea |
| F900 - FAFF | IsCJKCompatibilityIdeographs |
| FB00 - FB4F | IsAlphabeticPresentationForms |
| FB50 - FDFF | IsArabicPresentationForms-A |
| FE00 - FE0F | IsVariationSelectors |
| FE20 - FE2F | IsCombiningHalfMarks |
| FE30 - FE4F | IsCJKCompatibilityForms |
| FE50 - FE6F | IsSmallFormVariants |
| FE70 - FEFF | IsArabicPresentationForms-B |
| FF00 - FFFF | IsHalfwidthandFullwidthForms |
| FFFF - FFFF | IsSpecials |

Character class subtraction: [base_group - [excluded_group]]

A character class defines a set of characters. Character class subtraction yields a set of characters that is the result of excluding the characters in one character class from another character class.

A character class subtraction expression has the following form:

[*base_group* -[*excluded_group*]]

The square brackets ([]) and hyphen (-) are mandatory. The *base_group* is a [positive character group](#) or a [negative character group](#). The *excluded_group* component is another positive or negative character group, or another character class subtraction expression (that is, you can nest character class subtraction expressions).

For example, suppose you have a base group that consists of the character range from "a" through "z". To define the set of characters that consists of the base group except for the character "m", use `[a-z-[m]]`. To define the set of characters that consists of the base group except for the set of characters "d", "j", and "p", use `[a-z-[djp]]`. To define the set of characters that consists of the base group except for the character range from "m" through "p", use `[a-z-[m-p]]`.

Consider the nested character class subtraction expression, `[a-z-[d-w-[m-o]]]`. The expression is evaluated from the innermost character range outward. First, the character range from "m" through "o" is subtracted from the character range "d" through "w", which yields the set of characters from "d" through "l" and "p" through "w". That set is then subtracted from the character range from "a" through "z", which yields the set of characters `[abcuvwxyz]`.

You can use any character class with character class subtraction. To define the set of characters that consists of all Unicode characters from \u0000 through \uFFFF except white-space characters (`\s`), the characters in the punctuation general category (`\p{P}`), the characters in the `IsGreek` named block (`\p{IsGreek}`), and the Unicode NEXT LINE control character (`\x85`), use `[\u0000-\uFFFF-[\s\p{P}\p{IsGreek}\x85]]`.

Choose character classes for a character class subtraction expression that will yield useful results. Avoid an expression that yields an empty set of characters, which cannot match anything, or an expression that is equivalent to the original base group. For example, the empty set is the result of the expression `[\p{IsBasicLatin}-[\x00-\x7F]]`, which subtracts all characters in the `IsBasicLatin` character range from the `IsBasicLatin` general category. Similarly, the original base group is the result of the expression `[a-z-[0-9]]`. This is because the base group, which is the character range of letters from "a" through "z", does not contain any characters in the excluded group, which is the character range of decimal digits from "0" through "9".

The following example defines a regular expression, `^[0-9-[2468]]+$`, that matches zero and odd digits in an input string. The regular expression is interpreted as shown in the following table.

| Element | Description |
|----------------|---|
| <code>^</code> | Begin the match at the start of the input string. |

| Element | Description |
|---------------|--|
| [0-9-[2468]]+ | Match one or more occurrences of any character from 0 to 9 except for 2, 4, 6, and 8. In other words, match one or more occurrences of zero or an odd digit. |
| \$ | End the match at the end of the input string. |

C#

```
using System;
using System.Text.RegularExpressions;

public class Example
{
    public static void Main()
    {
        string[] inputs = { "123", "13579753", "3557798", "335599901" };
        string pattern = @"^-[0-9-[2468]]+$";

        foreach (string input in inputs)
        {
            Match match = Regex.Match(input, pattern);
            if (match.Success)
                Console.WriteLine(match.Value);
        }
    }
}
// The example displays the following output:
//      13579753
//      335599901
```

See also

- [GetUnicodeCategory](#)
- [Regular Expression Language - Quick Reference](#)
- [Regular Expression Options](#)

 Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

more information, see [our contributor guide](#).

Anchors in Regular Expressions

Article • 06/27/2023

Anchors, or atomic zero-width assertions, specify a position in the string where a match must occur. When you use an anchor in your search expression, the regular expression engine does not advance through the string or consume characters; it looks for a match in the specified position only. For example, `^` specifies that the match must start at the beginning of a line or string. Therefore, the regular expression `^http:` matches "http:" only when it occurs at the beginning of a line. The following table lists the anchors supported by the regular expressions in .NET.

| Anchor | Description |
|-----------------|---|
| <code>^</code> | By default, the match must occur at the beginning of the string; in multiline mode, it must occur at the beginning of the line. For more information, see Start of String or Line . |
| <code>\$</code> | By default, the match must occur at the end of the string or before <code>\n</code> at the end of the string; in multiline mode, it must occur at the end of the line or before <code>\n</code> at the end of the line. For more information, see End of String or Line . |
| <code>\A</code> | The match must occur at the beginning of the string only (no multiline support). For more information, see Start of String Only . |
| <code>\Z</code> | The match must occur at the end of the string, or before <code>\n</code> at the end of the string. For more information, see End of String or Before Ending Newline . |
| <code>\z</code> | The match must occur at the end of the string only. For more information, see End of String Only . |
| <code>\G</code> | The match must start at the position where the previous match ended, or if there was no previous match, at the position in the string where matching started. For more information, see Contiguous Matches . |
| <code>\b</code> | The match must occur on a word boundary. For more information, see Word Boundary . |
| <code>\B</code> | The match must not occur on a word boundary. For more information, see Non-Word Boundary . |

Start of String or Line: `^`

By default, the `^` anchor specifies that the following pattern must begin at the first character position of the string. If you use `^` with the `RegexOptions.Multiline` option (see [Regular Expression Options](#)), the match must occur at the beginning of each line.

The following example uses the `^` anchor in a regular expression that extracts information about the years during which some professional baseball teams existed. The example calls two overloads of the `Regex.Matches` method:

- The call to the `Matches(String, String)` overload finds only the first substring in the input string that matches the regular expression pattern.
- The call to the `Matches(String, String, RegexOptions)` overload with the `options` parameter set to `RegexOptions.Multiline` finds all five substrings.

C#

```
using System;
using System.Text.RegularExpressions;

public class Example
{
    public static void Main()
    {
        string input = "Brooklyn Dodgers, National League, 1911, 1912, 1932-
1957\n" +
                      "Chicago Cubs, National League, 1903-present\n" +
                      "Detroit Tigers, American League, 1901-present\n" +
                      "New York Giants, National League, 1885-1957\n" +
                      "Washington Senators, American League, 1901-1960\n";
        string pattern = @"^((\w+(\s?)){2,}),\s(\w+\s\w+),(\s\d{4}(-
(\d{4}|present))?,?)+" ;
        Match match;

        match = Regex.Match(input, pattern);
        while (match.Success)
        {
            Console.Write("The {0} played in the {1} in",
                         match.Groups[1].Value, match.Groups[4].Value);
            foreach (Capture capture in match.Groups[5].Captures)
                Console.Write(capture.Value);

            Console.WriteLine(".");
            match = match.NextMatch();
        }
        Console.WriteLine();

        match = Regex.Match(input, pattern, RegexOptions.Multiline);
        while (match.Success)
        {
            Console.Write("The {0} played in the {1} in",
                         match.Groups[1].Value, match.Groups[4].Value);
            foreach (Capture capture in match.Groups[5].Captures)
                Console.Write(capture.Value);

            Console.WriteLine(".");
            match = match.NextMatch();
        }
    }
}
```

```

        }
        Console.WriteLine();
    }
}

// The example displays the following output:
//   The Brooklyn Dodgers played in the National League in 1911, 1912,
1932-1957.
//
//   The Brooklyn Dodgers played in the National League in 1911, 1912,
1932-1957.
//   The Chicago Cubs played in the National League in 1903-present.
//   The Detroit Tigers played in the American League in 1901-present.
//   The New York Giants played in the National League in 1885-1957.
//   The Washington Senators played in the American League in 1901-1960.

```

The regular expression pattern `^((\w+(\s?)){2,}),\s(\w+\s\w+),(\s\d{4}(-\d{4}|present))?,?)+` is defined as shown in the following table.

| Pattern | Description |
|---|---|
| <code>^</code> | Begin the match at the beginning of the input string (or the beginning of the line if the method is called with the RegexOptions.Multiline option). |
| <code>((\w+(\s?)){2,},)</code> | Match one or more word characters followed either by zero or by one space at least two times. This is the first capturing group. This expression also defines a second and third capturing group: The second consists of the captured word, and the third consists of the captured white space. |
| <code>,\s</code> | Match a comma followed by a white-space character. |
| <code>(\w+\s\w+)</code> | Match one or more word characters followed by a space, followed by one or more word characters. This is the fourth capturing group. |
| <code>,</code> | Match a comma. |
| <code>\s\d{4}</code> | Match a space followed by four decimal digits. |
| <code>(-\d{4} present)?</code> | Match zero or one occurrence of a hyphen followed by four decimal digits or the string "present". This is the sixth capturing group. It also includes a seventh capturing group. |
| <code>,?</code> | Match zero or one occurrence of a comma. |
| <code>(\s\d{4}(-\d{4} present))?,?)+</code> | Match one or more occurrences of the following: a space, four decimal digits, zero or one occurrence of a hyphen followed by four decimal digits or the string "present", and zero or one comma. This is the fifth capturing group. |

End of String or Line: \$

The `$` anchor specifies that the preceding pattern must occur at the end of the input string, or before `\n` at the end of the input string.

If you use `$` with the [RegexOptions.Multiline](#) option, the match can also occur at the end of a line. Note that `$` is satisfied at `\n` but not at `\r\n` (the combination of carriage return and newline characters, or CR/LF). To handle the CR/LF character combination, include `\r?\$` in the regular expression pattern. Note that `\r?\$` will include any `\r` in the match.

The following example adds the `$` anchor to the regular expression pattern used in the example in the [Start of String or Line](#) section. When used with the original input string, which includes five lines of text, the [Regex.Matches\(String, String\)](#) method is unable to find a match, because the end of the first line does not match the `$` pattern. When the original input string is split into a string array, the [Regex.Matches\(String, String\)](#) method succeeds in matching each of the five lines. When the [Regex.Matches\(String, String, RegexOptions\)](#) method is called with the `options` parameter set to [RegexOptions.Multiline](#), no matches are found because the regular expression pattern does not account for the carriage return character `\r`. However, when the regular expression pattern is modified by replacing `$` with `\r?\$`, calling the [Regex.Matches\(String, String, RegexOptions\)](#) method with the `options` parameter set to [RegexOptions.Multiline](#) again finds five matches.

C#

```
using System;
using System.Text.RegularExpressions;

public class Example
{
    public static void Main()
    {
        string cr = Environment.NewLine;
        string input = "Brooklyn Dodgers, National League, 1911, 1912, 1932-
1957" + cr +
                      "Chicago Cubs, National League, 1903-present" + cr +
                      "Detroit Tigers, American League, 1901-present" + cr +
+
                      "New York Giants, National League, 1885-1957" + cr +
                      "Washington Senators, American League, 1901-1960" +
cr;
        Match match;

        string basePattern = @"^((\w+(\s?)){2,}),\s(\w+\s\w+),(\s\d{4}(-
(\d{4}|present))?,?)+";
```

```

        string pattern = basePattern + "$";
        Console.WriteLine("Attempting to match the entire input string:");
        match = Regex.Match(input, pattern);
        while (match.Success)
        {
            Console.Write("The {0} played in the {1} in",
                         match.Groups[1].Value, match.Groups[4].Value);
            foreach (Capture capture in match.Groups[5].Captures)
                Console.Write(capture.Value);

            Console.WriteLine(".");
            match = match.NextMatch();
        }
        Console.WriteLine();

        string[] teams = input.Split(new String[] { cr },
StringSplitOptions.RemoveEmptyEntries);
        Console.WriteLine("Attempting to match each element in a string
array:");
        foreach (string team in teams)
        {
            match = Regex.Match(team, pattern);
            if (match.Success)
            {
                Console.Write("The {0} played in the {1} in",
                             match.Groups[1].Value, match.Groups[4].Value);
                foreach (Capture capture in match.Groups[5].Captures)
                    Console.Write(capture.Value);
                Console.WriteLine(".");
            }
        }
        Console.WriteLine();

        Console.WriteLine("Attempting to match each line of an input string
with '$':");
        match = Regex.Match(input, pattern, RegexOptions.Multiline);
        while (match.Success)
        {
            Console.Write("The {0} played in the {1} in",
                         match.Groups[1].Value, match.Groups[4].Value);
            foreach (Capture capture in match.Groups[5].Captures)
                Console.Write(capture.Value);

            Console.WriteLine(".");
            match = match.NextMatch();
        }
        Console.WriteLine();

        pattern = basePattern + "\r?";
        Console.WriteLine(@"Attempting to match each line of an input string
with '\r?:');
        match = Regex.Match(input, pattern, RegexOptions.Multiline);
        while (match.Success)
        {
            Console.Write("The {0} played in the {1} in",

```

```

                match.Groups[1].Value, match.Groups[4].Value);
        foreach (Capture capture in match.Groups[5].Captures)
            Console.WriteLine(capture.Value);

        Console.WriteLine(".");
        match = match.NextMatch();
    }
    Console.WriteLine();
}
}

// The example displays the following output:
//   Attempting to match the entire input string:
//
//   Attempting to match each element in a string array:
//   The Brooklyn Dodgers played in the National League in 1911, 1912,
1932-1957.
//   The Chicago Cubs played in the National League in 1903-present.
//   The Detroit Tigers played in the American League in 1901-present.
//   The New York Giants played in the National League in 1885-1957.
//   The Washington Senators played in the American League in 1901-1960.
//
//   Attempting to match each line of an input string with '$':
//
//   Attempting to match each line of an input string with '\r?$':
//   The Brooklyn Dodgers played in the National League in 1911, 1912,
1932-1957.
//   The Chicago Cubs played in the National League in 1903-present.
//   The Detroit Tigers played in the American League in 1901-present.
//   The New York Giants played in the National League in 1885-1957.
//   The Washington Senators played in the American League in 1901-1960.

```

Start of String Only: \A

The `\A` anchor specifies that a match must occur at the beginning of the input string. It is identical to the `^` anchor, except that `\A` ignores the [RegexOptions.Multiline](#) option. Therefore, it can only match the start of the first line in a multiline input string.

The following example is similar to the examples for the `^` and `$` anchors. It uses the `\A` anchor in a regular expression that extracts information about the years during which some professional baseball teams existed. The input string includes five lines. The call to the [Regex.Matches\(String, String, RegexOptions\)](#) method finds only the first substring in the input string that matches the regular expression pattern. As the example shows, the [Multiline](#) option has no effect.

C#

```

using System;
using System.Text.RegularExpressions;

```

```

public class Example
{
    public static void Main()
    {
        string input = "Brooklyn Dodgers, National League, 1911, 1912, 1932-
1957\n" +
                      "Chicago Cubs, National League, 1903-present\n" +
                      "Detroit Tigers, American League, 1901-present\n" +
                      "New York Giants, National League, 1885-1957\n" +
                      "Washington Senators, American League, 1901-1960\n";

        string pattern = @"\A(((\w+(\s?)){2,}),\s(\w+\s\w+),(\s\d{4}(-
(\d{4}|present))?,?)+";

        Match match = Regex.Match(input, pattern, RegexOptions.Multiline);
        while (match.Success)
        {
            Console.Write("The {0} played in the {1} in",
                         match.Groups[1].Value, match.Groups[4].Value);
            foreach (Capture capture in match.Groups[5].Captures)
                Console.Write(capture.Value);

            Console.WriteLine(".");
            match = match.NextMatch();
        }
        Console.WriteLine();
    }
}

// The example displays the following output:
//   The Brooklyn Dodgers played in the National League in 1911, 1912,
//   1932-1957.

```

End of String or Before Ending Newline: \Z

The `\Z` anchor specifies that a match must occur at the end of the input string, or before `\n` at the end of the input string. It is identical to the `$` anchor, except that `\Z` ignores the `RegexOptions.Multiline` option. Therefore, in a multiline string, it can only be satisfied by the end of the last line, or the last line before `\n`.

Note that `\Z` is satisfied at `\n` but is not satisfied at `\r\n` (the CR/LF character combination). To treat CR/LF as if it were `\n`, include `\r?\Z` in the regular expression pattern. Note that this will make the `\r` part of the match.

The following example uses the `\Z` anchor in a regular expression that is similar to the example in the [Start of String or Line](#) section, which extracts information about the years during which some professional baseball teams existed. The subexpression `\r?\Z` in the regular expression `^((\w+(\s?)){2,}),\s(\w+\s\w+),(\s\d{4}(-(\d{4}|present))?,?)+\r?`

\z is satisfied at the end of a string, and also at the end of a string that ends with \n or \r\n. As a result, each element in the array matches the regular expression pattern.

```
C#  
  
using System;  
using System.Text.RegularExpressions;  
  
public class Example  
{  
    public static void Main()  
    {  
        string[] inputs = { "Brooklyn Dodgers, National League, 1911, 1912,  
1932-1957",  
                           "Chicago Cubs, National League, 1903-present" +  
Environment.NewLine,  
                           "Detroit Tigers, American League, 1901-present" +  
Regex.Unescape(@"\n"),  
                           "New York Giants, National League, 1885-1957",  
                           "Washington Senators, American League, 1901-1960"  
+ Environment.NewLine};  
        string pattern = @"^((\w+(\s?)){2,}),\s(\w+\s\w+),(\s\d{4}(-  
(\d{4}|present))?,?)+\r?\Z";  
  
        foreach (string input in inputs)  
        {  
            Console.WriteLine(Regex.Escape(input));  
            Match match = Regex.Match(input, pattern);  
            if (match.Success)  
                Console.WriteLine("    Match succeeded.");  
            else  
                Console.WriteLine("    Match failed.");  
        }  
    }  
    // The example displays the following output:  
    // Brooklyn\ Dodgers,\ National\ League,\ 1911,\ 1912,\ 1932-1957  
    //         Match succeeded.  
    // Chicago\ Cubs,\ National\ League,\ 1903-present\r\n  
    //         Match succeeded.  
    // Detroit\ Tigers,\ American\ League,\ 1901-present\r\n  
    //         Match succeeded.  
    // New\ York\ Giants,\ National\ League,\ 1885-1957  
    //         Match succeeded.  
    // Washington\ Senators,\ American\ League,\ 1901-1960\r\n  
    //         Match succeeded.
```

End of String Only: \z

The `\z` anchor specifies that a match must occur at the end of the input string. Like the `$` language element, `\z` ignores the `RegexOptions.Multiline` option. Unlike the `\z` language element, `\z` is not satisfied by a `\n` character at the end of a string. Therefore, it can only match the end of the input string.

The following example uses the `\z` anchor in a regular expression that is otherwise identical to the example in the previous section, which extracts information about the years during which some professional baseball teams existed. The example tries to match each of five elements in a string array with the regular expression pattern `^((\w+(\s?)){2,}),\s(\w+\s\w+),(\s\d{4}(-(\d{4}|present))?,?)+'\r?\z`. Two of the strings end with carriage return and line feed characters, one ends with a line feed character, and two end with neither a carriage return nor a line feed character. As the output shows, only the strings without a carriage return or line feed character match the pattern.

```
C#
```

```
using System;
using System.Text.RegularExpressions;

public class Example
{
    public static void Main()
    {
        string[] inputs = { "Brooklyn Dodgers, National League, 1911, 1912,
1932-1957",
                            "Chicago Cubs, National League, 1903-present" +
Environment.NewLine,
                            "Detroit Tigers, American League, 1901-present\n",
                            "New York Giants, National League, 1885-1957",
                            "Washington Senators, American League, 1901-1960"
+ Environment.NewLine };
        string pattern = @"^((\w+(\s?)){2,}),\s(\w+\s\w+),(\s\d{4}(-
(\d{4}|present))?,?)+'\r?\z";

        foreach (string input in inputs)
        {
            Console.WriteLine(Regex.Escape(input));
            Match match = Regex.Match(input, pattern);
            if (match.Success)
                Console.WriteLine("  Match succeeded.");
            else
                Console.WriteLine("  Match failed.");
        }
    }
// The example displays the following output:
//   Brooklyn\ Dodgers,\ National\ League,\ 1911,\ 1912,\ 1932-1957
//       Match succeeded.
//   Chicago\ Cubs,\ National\ League,\ 1903-present\r\n
//       Match failed.
```

```
// Detroit\ Tigers,\ American\ League,\ 1901-present\n
//      Match failed.
// New\ York\ Giants,\ National\ League,\ 1885-1957
//      Match succeeded.
// Washington\ Senators,\ American\ League,\ 1901-1960\r\n
//      Match failed.
```

Contiguous Matches: \G

The `\G` anchor specifies that a match must occur at the point where the previous match ended, or if there was no previous match, at the position in the string where matching started. When you use this anchor with the [Regex.Matches](#) or [Match.NextMatch](#) method, it ensures that all matches are contiguous.

Tip

Typically, you place a `\G` anchor at the left end of your pattern. In the uncommon case you're performing a right-to-left search, place the `\G` anchor at the right end of your pattern.

The following example uses a regular expression to extract the names of rodent species from a comma-delimited string.

```
C#  
  
using System;
using System.Text.RegularExpressions;  
  
public class Example
{
    public static void Main()
    {
        string input = "capybara,squirrel,chipmunk,porcupine,gopher," +
                      "beaver,groundhog,hamster,guinea pig,gerbil," +
                      "chinchilla,prairie dog,mouse,rat";
        string pattern = @"\G(\w+\s?\w*),?";
        Match match = Regex.Match(input, pattern);
        while (match.Success)
        {
            Console.WriteLine(match.Groups[1].Value);
            match = match.NextMatch();
        }
    }
}  
// The example displays the following output:  
//      capybara  
//      squirrel
```

```
//      chipmunk
//      porcupine
//      gopher
//      beaver
//      groundhog
//      hamster
//      guinea pig
//      gerbil
//      chinchilla
//      prairie dog
//      mouse
//      rat
```

The regular expression `\G(\w+\s?\w*),?` is interpreted as shown in the following table.

| Pattern | Description |
|--------------------------|---|
| <code>\G</code> | Begin where the last match ended. |
| <code>\w+</code> | Match one or more word characters. |
| <code>\s?</code> | Match zero or one space. |
| <code>\w*</code> | Match zero or more word characters. |
| <code>(\w+\s?\w*)</code> | Match one or more word characters followed by zero or one space, followed by zero or more word characters. This is the first capturing group. |
| <code>,?</code> | Match zero or one occurrence of a literal comma character. |

Word Boundary: \b

The `\b` anchor specifies that the match must occur on a boundary between a word character (the `\w` language element) and a non-word character (the `\W` language element). Word characters consist of alphanumeric characters and underscores; a non-word character is any character that is not alphanumeric or an underscore. (For more information, see [Character Classes](#).) The match may also occur on a word boundary at the beginning or end of the string.

The `\b` anchor is frequently used to ensure that a subexpression matches an entire word instead of just the beginning or end of a word. The regular expression `\bare\w*\b` in the following example illustrates this usage. It matches any word that begins with the substring "are". The output from the example also illustrates that `\b` matches both the beginning and the end of the input string.

```

using System;
using System.Text.RegularExpressions;

public class Example
{
    public static void Main()
    {
        string input = "area bare arena mare";
        string pattern = @"\bare\w*\b";
        Console.WriteLine("Words that begin with 'are':");
        foreach (Match match in Regex.Matches(input, pattern))
            Console.WriteLine("{0} found at position {1}",
                match.Value, match.Index);
    }
}
// The example displays the following output:
//     Words that begin with 'are':
//     'area' found at position 0
//     'arena' found at position 10

```

The regular expression pattern is interpreted as shown in the following table.

| Pattern | Description |
|---------|-------------------------------------|
| \b | Begin the match at a word boundary. |
| are | Match the substring "are". |
| \w* | Match zero or more word characters. |
| \b | End the match at a word boundary. |

Non-Word Boundary: \B

The `\B` anchor specifies that the match must not occur on a word boundary. It is the opposite of the `\b` anchor.

The following example uses the `\B` anchor to locate occurrences of the substring "qu" in a word. The regular expression pattern `\Bqu\w+` matches a substring that begins with a "qu" that does not start a word and that continues to the end of the word.

C#

```

using System;
using System.Text.RegularExpressions;

public class Example
{

```

```

public static void Main()
{
    string input = "equity queen equip acquaint quiet";
    string pattern = @"\Bqu\w+";
    foreach (Match match in Regex.Matches(input, pattern))
        Console.WriteLine("{0} found at position {1}",
                          match.Value, match.Index);
}
// The example displays the following output:
//      'quity' found at position 1
//      'quip' found at position 14
//      'quaint' found at position 21

```

The regular expression pattern is interpreted as shown in the following table.

| Pattern | Description |
|---------|--|
| \B | Do not begin the match at a word boundary. |
| qu | Match the substring "qu". |
| \w+ | Match one or more word characters. |

See also

- [Regular Expression Language - Quick Reference](#)
- [Regular Expression Options](#)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

Grouping constructs in regular expressions

Article • 05/08/2023

Grouping constructs delineate the subexpressions of a regular expression and capture the substrings of an input string. You can use grouping constructs to do the following:

- Match a subexpression that's repeated in the input string.
- Apply a quantifier to a subexpression that has multiple regular expression language elements. For more information about quantifiers, see [Quantifiers](#).
- Include a subexpression in the string that's returned by the [Regex.Replace](#) and [Match.Result](#) methods.
- Retrieve individual subexpressions from the [Match.Groups](#) property and process them separately from the matched text as a whole.

The following table lists the grouping constructs supported by the .NET regular expression engine and indicates whether they are capturing or noncapturing.

| Grouping construct | Capturing or noncapturing |
|---|---------------------------|
| Matched subexpressions | Capturing |
| Named matched subexpressions | Capturing |
| Balancing group definitions | Capturing |
| Noncapturing groups | Noncapturing |
| Group options | Noncapturing |
| Zero-width positive lookahead assertions | Noncapturing |
| Zero-width negative lookahead assertions | Noncapturing |
| Zero-width positive lookbehind assertions | Noncapturing |
| Zero-width negative lookbehind assertions | Noncapturing |
| Atomic groups | Noncapturing |

For information on groups and the regular expression object model, see [Grouping constructs and regular expression objects](#).

Matched subexpressions

The following grouping construct captures a matched subexpression:

(*subexpression*)

Here, *subexpression* is any valid regular expression pattern. Captures that use parentheses are numbered automatically from left to right based on the order of the opening parentheses in the regular expression, starting from 1. However, [named capture groups](#) are always ordered last, after non-named capture groups. The capture that's numbered 0 is the text matched by the entire regular expression pattern.

① Note

By default, the (*subexpression*) language element captures the matched subexpression. But if the `RegexOptions` parameter of a regular expression pattern matching method includes the `RegexOptions.ExplicitCapture` flag, or if the `n` option is applied to this subexpression (see [Group options](#) later in this article), the matched subexpression is not captured.

You can access captured groups in four ways:

- By using the backreference construct within the regular expression. The matched subexpression is referenced in the same regular expression by using the syntax `\number`, where *number* is the ordinal number of the captured subexpression.
- By using the named backreference construct within the regular expression. The matched subexpression is referenced in the same regular expression by using the syntax `\k<name>`, where *name* is the name of a capturing group, or `\k<number>`, where *number* is the ordinal number of a capturing group. A capturing group has a default name that is identical to its ordinal number. For more information, see [Named matched subexpressions](#) later in this topic.
- By using the `$number` replacement sequence in a `Regex.Replace` or `Match.Result` method call, where *number* is the ordinal number of the captured subexpression.
- Programmatically, by using the `GroupCollection` object returned by the `Match.Groups` property. The member at position zero in the collection represents the entire regular expression match. Each subsequent member represents a matched subexpression. For more information, see the [Grouping Constructs and Regular Expression Objects](#) section.

The following example illustrates a regular expression that identifies duplicated words in text. The regular expression pattern's two capturing groups represent the two instances

of the duplicated word. The second instance is captured to report its starting position in the input string.

```
C#  
  
using System;  
using System.Text.RegularExpressions;  
  
public class Example  
{  
    public static void Main()  
    {  
        string pattern = @"(\w+)\s(\1)\W";  
        string input = "He said that that was the the correct answer.";  
        foreach (Match match in Regex.Matches(input, pattern,  
RegexOptions.IgnoreCase))  
            Console.WriteLine("Duplicate '{0}' found at positions {1} and  
{2}.",  
                match.Groups[1].Value, match.Groups[1].Index,  
                match.Groups[2].Index);  
    }  
}  
// The example displays the following output:  
//     Duplicate 'that' found at positions 8 and 13.  
//     Duplicate 'the' found at positions 22 and 26.
```

The regular expression pattern is the following:

```
(\w+)\s(\1)\W
```

The following table shows how the regular expression pattern is interpreted.

| Pattern | Description |
|---------|---|
| (\w+) | Match one or more word characters. This is the first capturing group. |
| \s | Match a white-space character. |
| (\1) | Match the string in the first captured group. This is the second capturing group. The example assigns it to a captured group so that the starting position of the duplicate word can be retrieved from the <code>Match.Index</code> property. |
| \W | Match a non-word character, including white space and punctuation. This prevents the regular expression pattern from matching a word that starts with the word from the first captured group. |

Named matched subexpressions

The following grouping construct captures a matched subexpression and lets you access it by name or by number:

```
(?<name>subexpression)
```

or:

```
(?'name' subexpression)
```

Here, *name* is a valid group name, and *subexpression* is any valid regular expression pattern. *name* must not contain any punctuation characters and cannot begin with a number.

ⓘ Note

If the `RegexOptions` parameter of a regular expression pattern matching method includes the `RegexOptions.ExplicitCapture` flag, or if the `n` option is applied to this subexpression (see [Group options](#) later in this topic), the only way to capture a subexpression is to explicitly name capturing groups.

You can access named captured groups in the following ways:

- By using the named backreference construct within the regular expression. The matched subexpression is referenced in the same regular expression by using the syntax `\k<name>`, where *name* is the name of the captured subexpression.
- By using the backreference construct within the regular expression. The matched subexpression is referenced in the same regular expression by using the syntax `\number`, where *number* is the ordinal number of the captured subexpression. Named matched subexpressions are numbered consecutively from left to right after matched subexpressions.
- By using the `${name}` replacement sequence in a `Regex.Replace` or `Match.Result` method call, where *name* is the name of the captured subexpression.
- By using the `$number` replacement sequence in a `Regex.Replace` or `Match.Result` method call, where *number* is the ordinal number of the captured subexpression.
- Programmatically, by using the `GroupCollection` object returned by the `Match.Groups` property. The member at position zero in the collection represents the entire regular expression match. Each subsequent member represents a matched subexpression. Named captured groups are stored in the collection after numbered captured groups.

- Programmatically, by providing the subexpression name to the `GroupCollection` object's indexer (in C#) or to its `Item[]` property (in Visual Basic).

A simple regular expression pattern illustrates how numbered (unnamed) and named groups can be referenced either programmatically or by using regular expression language syntax. The regular expression `((?<One>abc)\d+)?(?<Two>xyz)(.*)` produces the following capturing groups by number and by name. The first capturing group (number 0) always refers to the entire pattern. (Named groups are always ordered last.)

| Number | Name | Pattern |
|--------|------------------|---|
| 0 | 0 (default name) | <code>((?<One>abc)\d+)?(?<Two>xyz)(.*)</code> |
| 1 | 1 (default name) | <code>((?<One>abc)\d+)</code> |
| 2 | 2 (default name) | <code>(.*)</code> |
| 3 | One | <code>(?<One>abc)</code> |
| 4 | Two | <code>(?<Two>xyz)</code> |

The following example illustrates a regular expression that identifies duplicated words and the word that immediately follows each duplicated word. The regular expression pattern defines two named subexpressions: `duplicateWord`, which represents the duplicated word, and `nextWord`, which represents the word that follows the duplicated word.

C#

```
using System;
using System.Text.RegularExpressions;

public class Example
{
    public static void Main()
    {
        string pattern = @"(?<duplicateWord>\w+)\s\k<duplicateWord>\W(?<nextWord>\w+)";
        string input = "He said that that was the the correct answer.";
        foreach (Match match in Regex.Matches(input, pattern,
RegexOptions.IgnoreCase))
            Console.WriteLine("A duplicate '{0}' at position {1} is followed by
'{2}'.",
                match.Groups["duplicateWord"].Value,
                match.Groups["duplicateWord"].Index,
                match.Groups["nextWord"].Value);
    }
}

// The example displays the following output:
```

```
//      A duplicate 'that' at position 8 is followed by 'was'.
//      A duplicate 'the' at position 22 is followed by 'correct'.
```

The regular expression pattern is as follows:

```
(?<duplicateWord>\w+)\s\k<duplicateWord>\W(?<nextWord>\w+)
```

The following table shows how the regular expression is interpreted.

| Pattern | Description |
|---------------------------|---|
| (? <duplicateWord>\w+) | Match one or more word characters. Name this capturing group <code>duplicateWord</code> . |
| \s | Match a white-space character. |
| \k<duplicateWord> | Match the string from the captured group that is named <code>duplicateWord</code> . |
| \W | Match a non-word character, including white space and punctuation. This prevents the regular expression pattern from matching a word that starts with the word from the first captured group. |
| (?<nextWord>\w+) | Match one or more word characters. Name this capturing group <code>nextWord</code> . |

A group name can be repeated in a regular expression. For example, it's possible for more than one group to be named `digit`, as the following example illustrates. In the case of duplicate names, the value of the [Group](#) object is determined by the last successful capture in the input string. In addition, the [CaptureCollection](#) is populated with information about each capture just as it would be if the group name was not duplicated.

In the following example, the regular expression `\D+(?<digit>\d+)\D+(?<digit>\d+)?` includes two occurrences of a group named `digit`. The first `digit` named group captures one or more digit characters. The second `digit` named group captures either zero or one occurrence of one or more digit characters. As the output from the example shows, if the second capturing group successfully matches text, the value of that text defines the value of the [Group](#) object. If the second capturing group does not match the input string, the value of the last successful match defines the value of the [Group](#) object.

```
C#
```

```
using System;
using System.Text.RegularExpressions;

public class Example
{
```

```

public static void Main()
{
    String pattern = @"\D+(?<digit>\d+)\D+(?<digit>\d+)?";
    String[] inputs = { "abc123def456", "abc123def" };
    foreach (var input in inputs) {
        Match m = Regex.Match(input, pattern);
        if (m.Success) {
            Console.WriteLine("Match: {0}", m.Value);
            for (int grpCtr = 1; grpCtr < m.Groups.Count; grpCtr++) {
                Group grp = m.Groups[grpCtr];
                Console.WriteLine("Group {0}: {1}", grpCtr, grp.Value);
                for (int capCtr = 0; capCtr < grp.Captures.Count; capCtr++)
                    Console.WriteLine("    Capture {0}: {1}",
                                      capCtr,
                                      grp.Captures[capCtr].Value);
            }
        }
        else {
            Console.WriteLine("The match failed.");
        }
        Console.WriteLine();
    }
}
// The example displays the following output:
//      Match: abc123def456
//          Group 1: 456
//              Capture 0: 123
//              Capture 1: 456
//
//      Match: abc123def
//          Group 1: 123
//              Capture 0: 123

```

The following table shows how the regular expression is interpreted.

| Pattern | Description |
|-------------------------|---|
| \D+ | Match one or more non-decimal digit characters. |
| (?<digit>\d+) | Match one or more decimal digit characters. Assign the match to the <code>digit</code> named group. |
| \D+ | Match one or more non-decimal digit characters. |
| (? <digit>\d+)?</digit> | Match zero or one occurrence of one or more decimal digit characters. Assign the match to the <code>digit</code> named group. |

Balancing group definitions

A balancing group definition deletes the definition of a previously defined group and stores, in the current group, the interval between the previously defined group and the current group. This grouping construct has the following format:

```
(?<name1-name2>subexpression)
```

or:

```
(?'name1-name2' subexpression)
```

Here, *name1* is the current group (optional), *name2* is a previously defined group, and *subexpression* is any valid regular expression pattern. The balancing group definition deletes the definition of *name2* and stores the interval between *name2* and *name1* in *name1*. If no *name2* group is defined, the match backtracks. Because deleting the last definition of *name2* reveals the previous definition of *name2*, this construct lets you use the stack of captures for group *name2* as a counter for keeping track of nested constructs such as parentheses or opening and closing brackets.

The balancing group definition uses *name2* as a stack. The beginning character of each nested construct is placed in the group and in its [Group.Captures](#) collection. When the closing character is matched, its corresponding opening character is removed from the group, and the [Captures](#) collection is decreased by one. After the opening and closing characters of all nested constructs have been matched, *name2* is empty.

Note

After you modify the regular expression in the following example to use the appropriate opening and closing character of a nested construct, you can use it to handle most nested constructs, such as mathematical expressions or lines of program code that include multiple nested method calls.

The following example uses a balancing group definition to match left and right angle brackets (<>) in an input string. The example defines two named groups, `open` and `close`, that are used like a stack to track matching pairs of angle brackets. Each captured left angle bracket is pushed into the capture collection of the `open` group, and each captured right angle bracket is pushed into the capture collection of the `close` group. The balancing group definition ensures that there is a matching right angle bracket for each left angle bracket. If there is not, the final subpattern, `(?(Open)(?!))`, is evaluated only if the `open` group is not empty (and, therefore, if all nested constructs have not been closed). If the final subpattern is evaluated, the match fails, because the `(?!)` subpattern is a zero-width negative lookahead assertion that always fails.

C#

```
using System;
using System.Text.RegularExpressions;

class Example
{
    public static void Main()
    {
        string pattern = "^[^<>]*" +
                         "(" +
                         "(?'Open'<)[^<>]*)+" +
                         "(?'Close-Open'>)[^<>]*)+" +
                         ")"* +
                         "(?(Open)(?!))$";
        string input = "<abc><mno<xyz>>";

        Match m = Regex.Match(input, pattern);
        if (m.Success == true)
        {
            Console.WriteLine("Input: \"{0}\" \nMatch: \"{1}\"", input, m);
            int grpCtr = 0;
            foreach (Group grp in m.Groups)
            {
                Console.WriteLine("    Group {0}: {1}", grpCtr, grp.Value);
                grpCtr++;
                int capCtr = 0;
                foreach (Capture cap in grp.Captures)
                {
                    Console.WriteLine("        Capture {0}: {1}", capCtr,
cap.Value);
                    capCtr++;
                }
            }
        }
        else
        {
            Console.WriteLine("Match failed.");
        }
    }
}

// The example displays the following output:
//   Input: "<abc><mno<xyz>>"
//   Match: "<abc><mno<xyz>>"
//       Group 0: <abc><mno<xyz>>
//           Capture 0: <abc><mno<xyz>>
//           Group 1: <mno<xyz>>
//               Capture 0: <abc>
//               Capture 1: <mno<xyz>>
//               Group 2: <xyz
//                   Capture 0: <abc
//                   Capture 1: <mno
//                   Capture 2: <xyz
//                   Group 3: >
```

```

//      Capture 0: >
//      Capture 1: >
//      Capture 2: >
//      Group 4:
//      Group 5: mno<xyz>
//      Capture 0: abc
//      Capture 1: xyz
//      Capture 2: mno<xyz>

```

The regular expression pattern is:

```
^[^<>]*(((?'Open'<)[^<>]*+)((?'Close-Open'>)[^<>]*+)*?(?Open)(?!))$
```

The regular expression is interpreted as follows:

| Pattern | Description |
|---|--|
| <code>^</code> | Begin at the start of the string. |
| <code>[^<>]*</code> | Match zero or more characters that are not left or right angle brackets. |
| <code>(?'Open'<)</code> | Match a left angle bracket and assign it to a group named <code>Open</code> . |
| <code>[^<>]*</code> | Match zero or more characters that are not left or right angle brackets. |
| <code>((?'Open'<)[^<>]*+)</code> | Match one or more occurrences of a left angle bracket followed by zero or more characters that are not left or right angle brackets. This is the second capturing group. |
| <code>(?'Close-Open'>)</code> | Match a right angle bracket, assign the substring between the <code>Open</code> group and the current group to the <code>Close</code> group, and delete the definition of the <code>Open</code> group. |
| <code>[^<>]*</code> | Match zero or more occurrences of any character that is neither a left nor a right angle bracket. |
| <code>((?'Close-Open'>)[^<>]*+)</code> | Match one or more occurrences of a right angle bracket, followed by zero or more occurrences of any character that is neither a left nor a right angle bracket. When matching the right angle bracket, assign the substring between the <code>Open</code> group and the current group to the <code>Close</code> group, and delete the definition of the <code>Open</code> group. This is the third capturing group. |
| <code>((?'Open'<)[^<>]*+((?'Close-Open'>)[^<>]*+)*?)</code> | Match zero or more occurrences of the following pattern: one or more occurrences of a left angle bracket, followed by zero or more non-angle bracket characters, followed by one or more occurrences of a right angle bracket, followed by zero or more occurrences of non-angle brackets. When matching the right angle bracket, delete the definition of the <code>Open</code> group, and assign the substring between the <code>Open</code> group and the current group to the <code>Close</code> group. This is the first capturing group. |

| Pattern | Description |
|---------------|--|
| (?(Open)(?!)) | If the <code>Open</code> group exists, abandon the match if an empty string can be matched, but do not advance the position of the regular expression engine in the string. This is a zero-width negative lookahead assertion. Because an empty string is always implicitly present in an input string, this match always fails. Failure of this match indicates that the angle brackets are not balanced. |
| \$ | Match the end of the input string. |

The final subexpression, `(?(Open)(?!))`, indicates whether the nesting constructs in the input string are properly balanced (for example, whether each left angle bracket is matched by a right angle bracket). It uses conditional matching based on a valid captured group; for more information, see [Alternation Constructs](#). If the `Open` group is defined, the regular expression engine attempts to match the subexpression `(?!)` in the input string. The `Open` group should be defined only if nesting constructs are unbalanced. Therefore, the pattern to be matched in the input string should be one that always causes the match to fail. In this case, `(?!)` is a zero-width negative lookahead assertion that always fails, because an empty string is always implicitly present at the next position in the input string.

In the example, the regular expression engine evaluates the input string "`<abc><mno<xyz>`" as shown in the following table.

| Step | Pattern | Result |
|------|---------------------------------------|--|
| 1 | <code>^</code> | Starts the match at the beginning of the input string |
| 2 | <code>[^<>]*</code> | Looks for non-angle bracket characters before the left angle bracket; finds no matches. |
| 3 | <code>((? 'Open' <)</code> | Matches the left angle bracket in " <code><abc></code> " and assigns it to the <code>Open</code> group. |
| 4 | <code>[^<>]*</code> | Matches "abc". |
| 5 | <code>) +</code> | " <code><abc</code> " is the value of the second captured group. The next character in the input string is not a left angle bracket, so the regular expression engine does not loop back to the <code>(? 'Open' <)[^<>]*</code> subpattern. |
| 6 | <code>((? 'Close- Open' >)</code> | Matches the right angle bracket in " <code><abc></code> ", assigns "abc", which is the substring between the <code>Open</code> group and the right angle bracket, to the <code>Close</code> group, and deletes the current value (" <code><</code> ") of the <code>Open</code> group, leaving it empty. |

| Step | Pattern | Result |
|------|-------------------|---|
| 7 | [^<>]* | Looks for non-angle bracket characters after the right angle bracket; finds no matches. |
| 8 |)+ | The value of the third captured group is ">". The next character in the input string is not a right angle bracket, so the regular expression engine does not loop back to the <code>(?'Close-Open') [^<>]*</code> subpattern. |
| 9 |)* | The value of the first captured group is "<abc>". The next character in the input string is a left angle bracket, so the regular expression engine loops back to the <code>((?'Open'<)</code> subpattern. |
| 10 | ((?'Open'<) | Matches the left angle bracket in "<mnō" and assigns it to the <code>Open</code> group. Its Group.Captures collection now has a single value, "<". |
| 11 | [^<>]* | Matches "mnō". |
| 12 |)+ | "<mnō" is the value of the second captured group. The next character in the input string is an left angle bracket, so the regular expression engine loops back to the <code>(?'Open'<)[^<>]*</code> subpattern. |
| 13 | ((?'Open'<) | Matches the left angle bracket in "<xyz>" and assigns it to the <code>Open</code> group. The Group.Captures collection of the <code>Open</code> group now includes two captures: the left angle bracket from "<mnō", and the left angle bracket from "<xyz>". |
| 14 | [^<>]* | Matches "xyz". |
| 15 |)+ | "<xyz" is the value of the second captured group. The next character in the input string is not a left angle bracket, so the regular expression engine does not loop back to the <code>(?'Open'<)[^<>]*</code> subpattern. |
| 16 | ((?'Close-Open'>) | Matches the right angle bracket in "<xyz>". "xyz", assigns the substring between the <code>Open</code> group and the right angle bracket to the <code>Close</code> group, and deletes the current value of the <code>Open</code> group. The value of the previous capture (the left angle bracket in "<mnō") becomes the current value of the <code>Open</code> group. The Captures collection of the <code>Open</code> group now includes a single capture, the left angle bracket from "<xyz>". |
| 17 | [^<>]* | Looks for non-angle bracket characters; finds no matches. |
| 18 |)+ | The value of the third captured group is ">". |

| Step | Pattern | Result |
|------|-----------------------------------|--|
| | | The next character in the input string is a right angle bracket, so the regular expression engine loops back to the <code>((?'Close-Open'>)[^<>]*)</code> subpattern. |
| 19 | <code>((?'Close-Open'>)</code> | Matches the final right angle bracket in "xyz>", assigns "mno<xyz>" (the substring between the <code>Open</code> group and the right angle bracket) to the <code>Close</code> group, and deletes the current value of the <code>Open</code> group. The <code>Open</code> group is now empty. |
| 20 | <code>[^<>]*</code> | Looks for non-angle bracket characters; finds no matches. |
| 21 | <code>)+</code> | The value of the third captured group is ">". |
| | | The next character in the input string is not a right angle bracket, so the regular expression engine does not loop back to the <code>((?'Close-Open'>)[^<>]*)</code> subpattern. |
| 22 | <code>)*</code> | The value of the first captured group is "<mno<xyz>>". |
| | | The next character in the input string is not a left angle bracket, so the regular expression engine does not loop back to the <code>((?'Open'<)</code> subpattern. |
| 23 | <code>(?(Open)(?!))</code> | The <code>Open</code> group is not defined, so no match is attempted. |
| 24 | <code>\$</code> | Matches the end of the input string. |

Noncapturing groups

The following grouping construct does not capture the substring that's matched by a subexpression:

`(?:subexpression)`

Here, *subexpression* is any valid regular expression pattern. The noncapturing group construct is typically used when a quantifier is applied to a group, but the substrings captured by the group are of no interest.

ⓘ Note

If a regular expression includes nested grouping constructs, an outer noncapturing group construct does not apply to the inner nested group constructs.

The following example illustrates a regular expression that includes noncapturing groups. Note that the output does not include any captured groups.

```
C#  
  
using System;  
using System.Text.RegularExpressions;  
  
public class Example  
{  
    public static void Main()  
    {  
        string pattern = @"(?:\b(?:\w+)\W*)+\.";  
        string input = "This is a short sentence.";  
        Match match = Regex.Match(input, pattern);  
        Console.WriteLine("Match: {0}", match.Value);  
        for (int ctr = 1; ctr < match.Groups.Count; ctr++)  
            Console.WriteLine("    Group {0}: {1}", ctr,  
match.Groups[ctr].Value);  
    }  
}  
// The example displays the following output:  
//      Match: This is a short sentence.
```

The regular expression `(?:\b(?:\w+)\W*)+\.` matches a sentence that is terminated by a period. Because the regular expression focuses on sentences and not on individual words, grouping constructs are used exclusively as quantifiers. The regular expression pattern is interpreted as shown in the following table.

| Pattern | Description |
|--------------------------------|--|
| <code>\b</code> | Begin the match at a word boundary. |
| <code>(?:\w+)</code> | Match one or more word characters. Do not assign the matched text to a captured group. |
| <code>\W*</code> | Match zero or more non-word characters. |
| <code>(?:\b(?:\w+)\W*)+</code> | Match the pattern of one or more word characters starting at a word boundary, followed by zero or more non-word characters, one or more times. Do not assign the matched text to a captured group. |
| <code>\.</code> | Match a period. |

Group options

The following grouping construct applies or disables the specified options within a subexpression:

```
(?imnsx-imnsx: subexpression )
```

Here, *subexpression* is any valid regular expression pattern. For example, `(?i-s:)` turns on case insensitivity and disables single-line mode. For more information about the inline options you can specify, see [Regular Expression Options](#).

ⓘ Note

You can specify options that apply to an entire regular expression rather than a subexpression by using a `System.Text.RegularExpressions.Regex` class constructor or a static method. You can also specify inline options that apply after a specific point in a regular expression by using the `(?imnsx-imnsx)` language construct.

The group options construct is not a capturing group. That is, although any portion of a string that is captured by *subexpression* is included in the match, it is not included in a captured group nor used to populate the `GroupCollection` object.

For example, the regular expression `\b(?ix: d \w+)\s` in the following example uses inline options in a grouping construct to enable case-insensitive matching and ignore pattern white space in identifying all words that begin with the letter "d". The regular expression is defined as shown in the following table.

| Pattern | Description |
|---------------------------|--|
| <code>\b</code> | Begin the match at a word boundary. |
| <code>(?ix: d \w+)</code> | Using case-insensitive matching and ignoring white space in this pattern, match a "d" followed by one or more word characters. |
| <code>\s</code> | Match a white-space character. |

C#

```
string pattern = @"\b(?ix: d \w+)\s";
string input = "Dogs are decidedly good pets.";

foreach (Match match in Regex.Matches(input, pattern))
    Console.WriteLine("{0}// found at index {1}.", match.Value,
match.Index);
// The example displays the following output:
//    'Dogs // found at index 0.
//    'decidedly // found at index 9.
```

Zero-width positive lookahead assertions

The following grouping construct defines a zero-width positive lookahead assertion:

```
(?= subexpression )
```

Here, *subexpression* is any regular expression pattern. For a match to be successful, the input string must match the regular expression pattern in *subexpression*, although the matched substring is not included in the match result. A zero-width positive lookahead assertion does not backtrack.

Typically, a zero-width positive lookahead assertion is found at the end of a regular expression pattern. It defines a substring that must be found at the end of a string for a match to occur but that should not be included in the match. It is also useful for preventing excessive backtracking. You can use a zero-width positive lookahead assertion to ensure that a particular captured group begins with text that matches a subset of the pattern defined for that captured group. For example, if a capturing group matches consecutive word characters, you can use a zero-width positive lookahead assertion to require that the first character be an alphabetical uppercase character.

The following example uses a zero-width positive lookahead assertion to match the word that precedes the verb "is" in the input string.

C#

```
using System;
using System.Text.RegularExpressions;

public class Example
{
    public static void Main()
    {
        string pattern = @"\b\w+(?=\\sis\\b)";
        string[] inputs = { "The dog is a Malamute.",
                            "The island has beautiful birds.",
                            "The pitch missed home plate.",
                            "Sunday is a weekend day." };

        foreach (string input in inputs)
        {
            Match match = Regex.Match(input, pattern);
            if (match.Success)
                Console.WriteLine("{0}' precedes 'is'.", match.Value);
            else
                Console.WriteLine("{0}' does not match the pattern.", input);
        }
    }
}

// The example displays the following output:
//      'dog' precedes 'is'.
//      'The island has beautiful birds.' does not match the pattern.
```

```
//      'The pitch missed home plate.' does not match the pattern.  
//      'Sunday' precedes 'is'.
```

The regular expression `\b\w+(?=\\sis\\b)` is interpreted as shown in the following table.

| Pattern | Description |
|--------------------------------|---|
| <code>\b</code> | Begin the match at a word boundary. |
| <code>\w+</code> | Match one or more word characters. |
| <code>(? =\\sis\\b)</code> | Determine whether the word characters are followed by a white-space character and the string "is", which ends on a word boundary. If so, the match is successful. |

Zero-width negative lookahead assertions

The following grouping construct defines a zero-width negative lookahead assertion:

```
(?! subexpression)
```

Here, *subexpression* is any regular expression pattern. For the match to be successful, the input string must not match the regular expression pattern in *subexpression*, although the matched string is not included in the match result.

A zero-width negative lookahead assertion is typically used either at the beginning or at the end of a regular expression. At the beginning of a regular expression, it can define a specific pattern that should not be matched when the beginning of the regular expression defines a similar but more general pattern to be matched. In this case, it is often used to limit backtracking. At the end of a regular expression, it can define a subexpression that cannot occur at the end of a match.

The following example defines a regular expression that uses a zero-width lookahead assertion at the beginning of the regular expression to match words that do not begin with "un".

C#

```
using System;  
using System.Text.RegularExpressions;  
  
public class Example  
{  
    public static void Main()  
    {  
        string pattern = @"\b(?!un)\w+\b";  
        string input = "unite one unethical ethics use untie ultimate";
```

```

        foreach (Match match in Regex.Matches(input, pattern,
RegexOptions.IgnoreCase))
            Console.WriteLine(match.Value);
    }
}

// The example displays the following output:
//      one
//      ethics
//      use
//      ultimate

```

The regular expression `\b(?!un)\w+\b` is interpreted as shown in the following table.

| Pattern | Description |
|---------------------|---|
| <code>\b</code> | Begin the match at a word boundary. |
| <code>(?!un)</code> | Determine whether the next two characters are "un". If they are not, a match is possible. |
| <code>\w+</code> | Match one or more word characters. |
| <code>\b</code> | End the match at a word boundary. |

The following example defines a regular expression that uses a zero-width lookahead assertion at the end of the regular expression to match words that do not end with a punctuation character.

C#

```

using System;
using System.Text.RegularExpressions;

public class Example
{
    public static void Main()
    {
        string pattern = @"\b\w+\b(?!\\p{P})";
        string input = "Disconnected, disjointed thoughts in a sentence
fragment.";
        foreach (Match match in Regex.Matches(input, pattern))
            Console.WriteLine(match.Value);
    }
}

// The example displays the following output:
//      disjointed
//      thoughts
//      in
//      a
//      sentence

```

The regular expression `\b\w+\b(?!\\p{P})` is interpreted as shown in the following table.

| Pattern | Description |
|--------------------|--|
| <code>\b</code> | Begin the match at a word boundary. |
| <code>\w+</code> | Match one or more word characters. |
| <code>\b</code> | End the match at a word boundary. |
| <code>\p{P}</code> | If the next character is not a punctuation symbol (such as a period or a comma), the match succeeds. |

Zero-width positive lookbehind assertions

The following grouping construct defines a zero-width positive lookbehind assertion:

```
(?<= subexpression )
```

Here, *subexpression* is any regular expression pattern. For a match to be successful, *subexpression* must occur at the input string to the left of the current position, although *subexpression* is not included in the match result. A zero-width positive lookbehind assertion does not backtrack.

Zero-width positive lookbehind assertions are typically used at the beginning of regular expressions. The pattern that they define is a precondition for a match, although it is not a part of the match result.

For example, the following example matches the last two digits of the year for the twenty first century (that is, it requires that the digits "20" precede the matched string).

```
C#  
  
using System;  
using System.Text.RegularExpressions;  
  
public class Example  
{  
    public static void Main()  
    {  
        string input = "2010 1999 1861 2140 2009";  
        string pattern = @"(?<=\b20)\d{2}\b";  
  
        foreach (Match match in Regex.Matches(input, pattern))  
            Console.WriteLine(match.Value);  
    }  
}  
// The example displays the following output:
```

```
//      10  
//      09
```

The regular expression pattern `(?<=\b20)\d{2}\b` is interpreted as shown in the following table.

| Pattern | Description |
|---------------------------|--|
| <code>\d{2}</code> | Match two decimal digits. |
| <code>(?<=\b20)</code> | Continue the match if the two decimal digits are preceded by the decimal digits "20" on a word boundary. |
| <code>\b</code> | End the match at a word boundary. |

Zero-width positive lookbehind assertions are also used to limit backtracking when the last character or characters in a captured group must be a subset of the characters that match that group's regular expression pattern. For example, if a group captures all consecutive word characters, you can use a zero-width positive lookbehind assertion to require that the last character be alphabetical.

Zero-width negative lookbehind assertions

The following grouping construct defines a zero-width negative lookbehind assertion:

```
(?<! subexpression )
```

Here, *subexpression* is any regular expression pattern. For a match to be successful, *subexpression* must not occur at the input string to the left of the current position. However, any substring that does not match `subexpression` is not included in the match result.

Zero-width negative lookbehind assertions are typically used at the beginning of regular expressions. The pattern that they define precludes a match in the string that follows. They are also used to limit backtracking when the last character or characters in a captured group must not be one or more of the characters that match that group's regular expression pattern. For example, if a group captures all consecutive word characters, you can use a zero-width positive lookbehind assertion to require that the last character not be an underscore (_).

The following example matches the date for any day of the week that is not a weekend (that is, that is neither Saturday nor Sunday).

```

using System;
using System.Text.RegularExpressions;

public class Example
{
    public static void Main()
    {
        string[] dates = { "Monday February 1, 2010",
                           "Wednesday February 3, 2010",
                           "Saturday February 6, 2010",
                           "Sunday February 7, 2010",
                           "Monday, February 8, 2010" };
        string pattern = @"(?<!(Saturday|Sunday) )\b\w+ \d{1,2}, \d{4}\b";

        foreach (string dateValue in dates)
        {
            Match match = Regex.Match(dateValue, pattern);
            if (match.Success)
                Console.WriteLine(match.Value);
        }
    }
}

// The example displays the following output:
//      February 1, 2010
//      February 3, 2010
//      February 8, 2010

```

The regular expression pattern `(?<!(Saturday|Sunday))\b\w+ \d{1,2}, \d{4}\b` is interpreted as shown in the following table.

| Pattern | Description |
|--|---|
| <code>\b</code> | Begin the match at a word boundary. |
| <code>\w+</code> | Match one or more word characters followed by a white-space character. |
| <code>\d{1,2},</code> | Match either one or two decimal digits followed by a white-space character and a comma. |
| <code>\d{4}\b</code> | Match four decimal digits, and end the match at a word boundary. |
| <code>(?<!</code> <code>(Saturday Sunday))</code> | If the match is preceded by something other than the strings "Saturday" or "Sunday" followed by a space, the match is successful. |

Atomic groups

The following grouping construct represents an atomic group (known in some other regular expression engines as a nonbacktracking subexpression, an atomic

subexpression, or a once-only subexpression):

```
(?> subexpression )
```

Here, *subexpression* is any regular expression pattern.

Ordinarily, if a regular expression includes an optional or alternative matching pattern and a match does not succeed, the regular expression engine can branch in multiple directions to match an input string with a pattern. If a match is not found when it takes the first branch, the regular expression engine can back up or backtrack to the point where it took the first match and attempt the match using the second branch. This process can continue until all branches have been tried.

The `(?>subexpression)` language construct disables backtracking. The regular expression engine will match as many characters in the input string as it can. When no further match is possible, it will not backtrack to attempt alternate pattern matches. (That is, the subexpression matches only strings that would be matched by the subexpression alone; it does not attempt to match a string based on the subexpression and any subexpressions that follow it.)

This option is recommended if you know that backtracking will not succeed. Preventing the regular expression engine from performing unnecessary searching improves performance.

The following example illustrates how an atomic group modifies the results of a pattern match. The backtracking regular expression successfully matches a series of repeated characters followed by one more occurrence of the same character on a word boundary, but the nonbacktracking regular expression does not.

C#

```
using System;
using System.Text.RegularExpressions;

public class Example
{
    public static void Main()
    {
        string[] inputs = { "cccd.", "aaad", "aaaa" };
        string back = @"(\w)\1+\.\b";
        string noback = @"(?>(\w)\1+)\.\b";

        foreach (string input in inputs)
        {
            Match match1 = Regex.Match(input, back);
            Match match2 = Regex.Match(input, noback);
            Console.WriteLine("{0}: ", input);
```

```

        Console.Write("  Backtracking : ");
        if (match1.Success)
            Console.WriteLine(match1.Value);
        else
            Console.WriteLine("No match");

        Console.Write("  Nonbacktracking: ");
        if (match2.Success)
            Console.WriteLine(match2.Value);
        else
            Console.WriteLine("No match");
    }
}

// The example displays the following output:
//   cccd.:
//       Backtracking : cccd
//       Nonbacktracking: cccd
//   aaad:
//       Backtracking : aaad
//       Nonbacktracking: aaad
//   aaaa:
//       Backtracking : aaaa
//       Nonbacktracking: No match

```

The nonbacktracking regular expression `(?>(\w)\1+).\b` is defined as shown in the following table.

| Pattern | Description |
|-----------------------------|---|
| <code>(\w)</code> | Match a single word character and assign it to the first capturing group. |
| <code>\1+</code> | Match the value of the first captured substring one or more times. |
| <code>.</code> | Match any character. |
| <code>\b</code> | End the match on a word boundary. |
| <code>(?>(\w)\1+)</code> | Match one or more occurrences of a duplicated word character, but don't backtrack to match the last character on a word boundary. |

Grouping constructs and regular expression objects

Substrings that are matched by a regular expression capturing group are represented by [System.Text.RegularExpressions.Group](#) objects, which can be retrieved from the

`System.Text.RegularExpressions.GroupCollection` object that is returned by the `Match.Groups` property. The `GroupCollection` object is populated as follows:

- The first `Group` object in the collection (the object at index zero) represents the entire match.
- The next set of `Group` objects represent unnamed (numbered) capturing groups. They appear in the order in which they are defined in the regular expression, from left to right. The index values of these groups range from 1 to the number of unnamed capturing groups in the collection. (The index of a particular group is equivalent to its numbered backreference. For more information about backreferences, see [Backreference Constructs](#).)
- The final set of `Group` objects represent named capturing groups. They appear in the order in which they are defined in the regular expression, from left to right. The index value of the first named capturing group is one greater than the index of the last unnamed capturing group. If there are no unnamed capturing groups in the regular expression, the index value of the first named capturing group is one.

If you apply a quantifier to a capturing group, the corresponding `Group` object's `Capture.Value`, `Capture.Index`, and `Capture.Length` properties reflect the last substring that is captured by a capturing group. You can retrieve a complete set of substrings that are captured by groups that have quantifiers from the `CaptureCollection` object that is returned by the `Group.Captures` property.

The following example clarifies the relationship between the `Group` and `Capture` objects.

C#

```
using System;
using System.Text.RegularExpressions;

public class Example
{
    public static void Main()
    {
        string pattern = @"(\b(\w+)\W+)+";
        string input = "This is a short sentence.";
        Match match = Regex.Match(input, pattern);
        Console.WriteLine("Match: '{0}'", match.Value);
        for (int ctr = 1; ctr < match.Groups.Count; ctr++)
        {
            Console.WriteLine("    Group {0}: '{1}'", ctr,
match.Groups[ctr].Value);
            int capCtr = 0;
            foreach (Capture capture in match.Groups[ctr].Captures)
            {
                Console.WriteLine("        Capture {0}: '{1}'", capCtr,
capture.Value);
            }
        }
    }
}
```

```

        capCtr++;
    }
}
}

// The example displays the following output:
//      Match: 'This is a short sentence.'
//          Group 1: 'sentence.'
//              Capture 0: 'This '
//              Capture 1: 'is '
//              Capture 2: 'a '
//              Capture 3: 'short '
//              Capture 4: 'sentence.'
//          Group 2: 'sentence'
//              Capture 0: 'This'
//              Capture 1: 'is'
//              Capture 2: 'a'
//              Capture 3: 'short'
//              Capture 4: 'sentence'

```

The regular expression pattern `(\b(\w+)\W+)+` extracts individual words from a string. It is defined as shown in the following table.

| Pattern | Description |
|---------------------------|--|
| <code>\b</code> | Begin the match at a word boundary. |
| <code>(\w+)</code> | Match one or more word characters. Together, these characters form a word. This is the second capturing group. |
| <code>\W+</code> | Match one or more non-word characters. |
| <code>(\b(\w+)\W+)</code> | Match the pattern of one or more word characters followed by one or more non-word characters one or more times. This is the first capturing group. |

The second capturing group matches each word of the sentence. The first capturing group matches each word along with the punctuation and white space that follow the word. The [Group](#) object whose index is 2 provides information about the text matched by the second capturing group. The complete set of words captured by the capturing group are available from the [CaptureCollection](#) object returned by the [Group.Captures](#) property.

See also

- [Regular Expression Language - Quick Reference](#)
- [Backtracking](#)

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

Quantifiers in Regular Expressions

Article • 08/12/2022

Quantifiers specify how many instances of a character, group, or character class must be present in the input for a match to be found. The following table lists the quantifiers supported by .NET:

| Greedy quantifier | Lazy quantifier | Description |
|-------------------|-----------------|--------------------------------|
| * | *? | Matches zero or more times. |
| + | +? | Matches one or more times. |
| ? | ?? | Matches zero or one time. |
| { n } | { n }? | Matches exactly n times. |
| { n , } | { n , }? | Matches at least n times. |
| { n , m } | { n , m }? | Matches from n to m times. |

The quantities `n` and `m` are integer constants. Ordinarily, quantifiers are greedy. They cause the regular expression engine to match as many occurrences of particular patterns as possible. Appending the `?` character to a quantifier makes it lazy. It causes the regular expression engine to match as few occurrences as possible. For a complete description of the difference between greedy and lazy quantifiers, see the section [Greedy and Lazy Quantifiers](#) later in this article.

ⓘ Important

Nesting quantifiers, such as the regular expression pattern `(a*)*`, can increase the number of comparisons that the regular expression engine must perform. The number of comparisons can increase as an exponential function of the number of characters in the input string. For more information about this behavior and its workarounds, see [Backtracking](#).

Regular Expression Quantifiers

The following sections list the quantifiers supported by .NET regular expressions:

ⓘ Note

If the *, +, ?, {, and } characters are encountered in a regular expression pattern, the regular expression engine interprets them as quantifiers or part of quantifier constructs unless they are included in a **character class**. To interpret these as literal characters outside a character class, you must escape them by preceding them with a backslash. For example, the string * in a regular expression pattern is interpreted as a literal asterisk ("*") character.

Match Zero or More Times: *

The * quantifier matches the preceding element zero or more times. It's equivalent to the {0,} quantifier. * is a greedy quantifier whose lazy equivalent is *?.

The following example illustrates this regular expression. Five of the nine digit-groups in the input string match the pattern and four (95, 929, 9219, and 9919) don't.

C#

```
string pattern = @"\b91*9*\b";
string input = "99 95 919 929 9119 9219 999 9919 91119";
foreach (Match match in Regex.Matches(input, pattern))
    Console.WriteLine("{0}' found at position {1}.", match.Value,
match.Index);

// The example displays the following output:
//      '99' found at position 0.
//      '919' found at position 6.
//      '9119' found at position 14.
//      '999' found at position 24.
//      '91119' found at position 33.
```

The regular expression pattern is defined as shown in the following table:

| Pattern | Description |
|---------|---|
| \b | Specifies that the match must start at a word boundary. |
| 91* | Matches a 9 followed by zero or more 1 characters. |
| 9* | Matches zero or more 9 characters. |
| \b | Specifies that the match must end at a word boundary. |

Match One or More Times: +

The `+` quantifier matches the preceding element one or more times. It's equivalent to `{1,}`. `+` is a greedy quantifier whose lazy equivalent is `+?`.

For example, the regular expression `\ban+\w*?\b` tries to match entire words that begin with the letter `a` followed by one or more instances of the letter `n`. The following example illustrates this regular expression. The regular expression matches the words `an`, `annual`, `announcement`, and `antique`, and correctly fails to match `autumn` and `all`.

C#

```
string pattern = @"\ban+\w*?\b";

string input = "Autumn is a great time for an annual announcement to all
antique collectors.";
foreach (Match match in Regex.Matches(input, pattern,
RegexOptions.IgnoreCase))
    Console.WriteLine("{0}' found at position {1}.", match.Value,
match.Index);

// The example displays the following output:
//      'an' found at position 27.
//      'annual' found at position 30.
//      'announcement' found at position 37.
//      'antique' found at position 57.
```

The regular expression pattern is defined as shown in the following table:

| Pattern | Description |
|-------------------|--|
| <code>\b</code> | Start at a word boundary. |
| <code>an+</code> | Matches an <code>a</code> followed by one or more <code>n</code> characters. |
| <code>\w*?</code> | Matches a word character zero or more times but as few times as possible. |
| <code>\b</code> | End at a word boundary. |

Match Zero or One Time: ?

The `?` quantifier matches the preceding element zero or one time. It's equivalent to `{0,1}`. `?` is a greedy quantifier whose lazy equivalent is `??`.

For example, the regular expression `\ban?\b` tries to match entire words that begin with the letter `a` followed by zero or one instance of the letter `n`. In other words, it tries to match the words `a` and `an`. The following example illustrates this regular expression:

C#

```
string pattern = @"\ban?\b";
string input = "An amiable animal with a large snout and an animated nose.";
foreach (Match match in Regex.Matches(input, pattern,
RegexOptions.IgnoreCase))
    Console.WriteLine("{0}' found at position {1}.", match.Value,
match.Index);

// The example displays the following output:
//      'An' found at position 0.
//      'a' found at position 23.
//      'an' found at position 42.
```

The regular expression pattern is defined as shown in the following table:

| Pattern | Description |
|---------|---|
| \b | Start at a word boundary. |
| an? | Matches an a followed by zero or one n character. |
| \b | End at a word boundary. |

Match Exactly n Times: {n}

The {n} quantifier matches the preceding element exactly *n* times, where *n* is any integer. {n} is a greedy quantifier whose lazy equivalent is {n}?.

For example, the regular expression \b\d+\,\d{3}\b tries to match a word boundary followed by one or more decimal digits followed by three decimal digits followed by a word boundary. The following example illustrates this regular expression:

C#

```
string pattern = @"\b\d+\,\d{3}\b";
string input = "Sales totaled 103,524 million in January, " +
              "106,971 million in February, but only " +
              "943 million in March.";
foreach (Match match in Regex.Matches(input, pattern))
    Console.WriteLine("{0}' found at position {1}.", match.Value,
match.Index);

// The example displays the following output:
//      '103,524' found at position 14.
//      '106,971' found at position 45.
```

The regular expression pattern is defined as shown in the following table:

| Pattern | Description |
|---------|-------------------------------------|
| \b | Start at a word boundary. |
| \d+ | Matches one or more decimal digits. |
| \, | Matches a comma character. |
| \d{3} | Matches three decimal digits. |
| \b | End at a word boundary. |

Match at Least n Times: {n,}

The `{n,}` quantifier matches the preceding element at least n times, where n is any integer. `{n,}` is a greedy quantifier whose lazy equivalent is `{n,}?`.

For example, the regular expression `\b\d{2,}\b\D+` tries to match a word boundary followed by at least two digits followed by a word boundary and a non-digit character. The following example illustrates this regular expression. The regular expression fails to match the phrase `"7 days"` because it contains just one decimal digit, but it successfully matches the phrases `"10 weeks"` and `"300 years"`.

C#

```
string pattern = @"\b\d{2,}\b\D+";
string input = "7 days, 10 weeks, 300 years";
foreach (Match match in Regex.Matches(input, pattern))
    Console.WriteLine("{0}' found at position {1}.", match.Value,
match.Index);

// The example displays the following output:
//      '10 weeks, ' found at position 8.
//      '300 years' found at position 18.
```

The regular expression pattern is defined as shown in the following table:

| Pattern | Description |
|---------|--------------------------------------|
| \b | Start at a word boundary. |
| \d{2,} | Matches at least two decimal digits. |
| \b | Matches a word boundary. |

| Pattern | Description |
|---------|---|
| \D+ | Matches at least one non-decimal digit. |

Match Between n and m Times: {n,m}

The `{n,m}` quantifier matches the preceding element at least n times, but no more than m times, where n and m are integers. `{n,m}` is a greedy quantifier whose lazy equivalent is `{n,m}?`.

In the following example, the regular expression `(00\s){2,4}` tries to match between two and four occurrences of two zero digits followed by a space. The final portion of the input string includes this pattern five times rather than the maximum of four. However, only the initial portion of this substring (up to the space and the fifth pair of zeros) matches the regular expression pattern.

C#

```
string pattern = @"(00\s){2,4}";
string input = "0x00 FF 00 00 18 17 FF 00 00 00 21 00 00 00 00 00 00";
foreach (Match match in Regex.Matches(input, pattern))
    Console.WriteLine("{0} found at position {1}.", match.Value,
match.Index);

// The example displays the following output:
//      '00 00 ' found at position 8.
//      '00 00 00 ' found at position 23.
//      '00 00 00 00 ' found at position 35.
```

Match Zero or More Times (Lazy Match): *?

The `*?` quantifier matches the preceding element zero or more times but as few times as possible. It's the lazy counterpart of the greedy quantifier `*`.

In the following example, the regular expression `\b\w*?oo\w*?\b` matches all words that contain the string `oo`.

C#

```
string pattern = @"\b\w*?oo\w*?\b";
string input = "woof root root rob oof woo woe";
foreach (Match match in Regex.Matches(input, pattern,
RegexOptions.IgnoreCase))
    Console.WriteLine("{0} found at position {1}.", match.Value,
match.Index);
```

```
// The example displays the following output:
//      'woof' found at position 0.
//      'root' found at position 5.
//      'root' found at position 10.
//      'oof' found at position 19.
//      'woo' found at position 23.
```

The regular expression pattern is defined as shown in the following table:

| Pattern | Description |
|---------|---|
| \b | Start at a word boundary. |
| \w*? | Matches zero or more word characters but as few characters as possible. |
| oo | Matches the string oo. |
| \w*? | Matches zero or more word characters but as few characters as possible. |
| \b | End on a word boundary. |

Match One or More Times (Lazy Match): +?

The +? quantifier matches the preceding element one or more times but as few times as possible. It's the lazy counterpart of the greedy quantifier +.

For example, the regular expression \b\w+?\b matches one or more characters separated by word boundaries. The following example illustrates this regular expression:

C#

```
string pattern = @"\b\w+?\b";
string input = "Aa Bb Cc Dd Ee Ff";
foreach (Match match in Regex.Matches(input, pattern))
    Console.WriteLine("{0} found at position {1}.", match.Value,
match.Index);

// The example displays the following output:
//      'Aa' found at position 0.
//      'Bb' found at position 3.
//      'Cc' found at position 6.
//      'Dd' found at position 9.
//      'Ee' found at position 12.
//      'Ff' found at position 15.
```

Match Zero or One Time (Lazy Match): ??

The `??` quantifier matches the preceding element zero or one time but as few times as possible. It's the lazy counterpart of the greedy quantifier `?`.

For example, the regular expression `^\s*(System.)??Console.WriteLine(?)\(\??` attempts to match the strings `Console.Write` or `Console.WriteLine`. The string can also include `System.` before `Console`, and it can be followed by an opening parenthesis. The string must be at the beginning of a line, although it can be preceded by white space. The following example illustrates this regular expression:

C#

```
string pattern = @"^\s*(System.)??Console.WriteLine(?)\(\??";
string input = "System.Console.WriteLine(\"Hello!\")\n" +
               "Console.Write(\"Hello!\")\n" +
               "Console.WriteLine(\"Hello!\")\n" +
               "Console.ReadLine()\n" +
               "    Console.WriteLine";
foreach (Match match in Regex.Matches(input, pattern,
                                         RegexOptions.IgnorePatternWhitespace |
                                         RegexOptions.IgnoreCase |
                                         RegexOptions.Multiline))
    Console.WriteLine("{0}' found at position {1}.", match.Value,
                      match.Index);

// The example displays the following output:
//      'System.Console.WriteLine' found at position 0.
//      'Console.WriteLine' found at position 36.
//      'Console.ReadLine' found at position 61.
//      '    Console.WriteLine' found at position 110.
```

The regular expression pattern is defined as shown in the following table:

| Pattern | Description |
|-----------------------------------|---|
| <code>^</code> | Matches the start of the input stream. |
| <code>\s*</code> | Matches zero or more white-space characters. |
| <code>(System.)??</code> | Matches zero or one occurrence of the string <code>System.</code> . |
| <code>Console.WriteLine(?)</code> | Matches the string <code>Console.WriteLine</code> . |
| <code>\(\??</code> | Matches zero or one occurrence of the opening parenthesis. |

Match Exactly n Times (Lazy Match): {n}?

The `{n}?` quantifier matches the preceding element exactly `n` times, where `n` is any integer. It's the lazy counterpart of the greedy quantifier `{n}`.

In the following example, the regular expression `\b(\w{3,}?\.){2}?\w{3,}?\b` is used to identify a website address. The expression matches `www.microsoft.com` and `msdn.microsoft.com` but doesn't match `mywebsite` or `mycompany.com`.

C#

```
string pattern = @"\b(\w{3,}?\.){2}?\w{3,}?\b";
string input = "www.microsoft.com msdn.microsoft.com mywebsite
mycompany.com";
foreach (Match match in Regex.Matches(input, pattern))
    Console.WriteLine("{0}' found at position {1}.", match.Value,
match.Index);

// The example displays the following output:
//      'www.microsoft.com' found at position 0.
//      'msdn.microsoft.com' found at position 18.
```

The regular expression pattern is defined as shown in the following table:

| Pattern | Description |
|--------------------------|---|
| <code>\b</code> | Start at a word boundary. |
| <code>(\w{3,}?\.)</code> | Matches at least three word-characters but as few characters as possible, followed by a dot or period character. This pattern is the first capturing group. |
| <code>{2}?</code> | Matches the pattern in the first group two times but as few times as possible. |
| <code>\b</code> | End the match on a word boundary. |

Match at Least n Times (Lazy Match): `{n,}?`

The `{n,}?` quantifier matches the preceding element at least `n` times, where `n` is any integer but as few times as possible. It's the lazy counterpart of the greedy quantifier `{n,}`.

See the example for the `{n}?` quantifier in the previous section for an illustration. The regular expression in that example uses the `{n,}?` quantifier to match a string that has at least three characters followed by a period.

Match Between n and m Times (Lazy Match): `{n,m}?`

The `{n,m}?` quantifier matches the preceding element between `n` and `m` times, where `n` and `m` are integers but as few times as possible. It's the lazy counterpart of the greedy quantifier `{n,m}`.

In the following example, the regular expression `\b[A-Z](\w*\?\s*){1,10}[.!\?]` matches sentences that contain between 1 and 10 words. It matches all the sentences in the input string except for one sentence that contains 18 words.

C#

```
string pattern = @"\b[A-Z](\w*\?\s*){1,10}[.!\?]";
string input = "Hi. I am writing a short note. Its purpose is " +
                "to test a regular expression that attempts to find " +
                "sentences with ten or fewer words. Most sentences " +
                "in this note are short.";
foreach (Match match in Regex.Matches(input, pattern))
    Console.WriteLine("{0} found at position {1}.", match.Value,
match.Index);

// The example displays the following output:
//      'Hi.' found at position 0.
//      'I am writing a short note.' found at position 4.
//      'Most sentences in this note are short.' found at position 132.
```

The regular expression pattern is defined as shown in the following table:

| Pattern | Description |
|------------------------|---|
| <code>\b</code> | Start at a word boundary. |
| <code>[A-Z]</code> | Matches an uppercase character from A to Z. |
| <code>(\w*?\s*)</code> | Matches zero or more word characters, followed by one or more white-space characters but as few times as possible. This pattern is the first capturing group. |
| <code>{1,10}</code> | Matches the previous pattern between 1 and 10 times. |
| <code>[.!\?]</code> | Matches any one of the punctuation characters <code>.</code> , <code>!</code> , or <code>?</code> . |

Greedy and Lazy Quantifiers

Some quantifiers have two versions:

- A greedy version.

A greedy quantifier tries to match an element as many times as possible.

- A non-greedy (or lazy) version.

A non-greedy quantifier tries to match an element as few times as possible. You can turn a greedy quantifier into a lazy quantifier by adding a `?`.

Consider a regular expression that's intended to extract the last four digits from a string of numbers, such as a credit card number. The version of the regular expression that uses the `*` greedy quantifier is `\b.*([0-9]{4})\b`. However, if a string contains two numbers, this regular expression matches the last four digits of the second number only, as the following example shows:

```
C#  
  
string greedyPattern = @"\b.*([0-9]{4})\b";  
string input1 = "1112223333 3992991999";  
foreach (Match match in Regex.Matches(input1, greedyPattern))  
    Console.WriteLine("Account ending in *****{0}.", match.Groups[1].Value);  
  
// The example displays the following output:  
//      Account ending in *****1999.
```

The regular expression fails to match the first number because the `*` quantifier tries to match the previous element as many times as possible in the entire string, and so it finds its match at the end of the string.

This behavior isn't the desired one. Instead, you can use the `*?` lazy quantifier to extract digits from both numbers, as the following example shows:

```
C#  
  
string lazyPattern = @"\b.*?([0-9]{4})\b";  
string input2 = "1112223333 3992991999";  
foreach (Match match in Regex.Matches(input2, lazyPattern))  
    Console.WriteLine("Account ending in *****{0}.", match.Groups[1].Value);  
  
// The example displays the following output:  
//      Account ending in *****3333.  
//      Account ending in *****1999.
```

In most cases, regular expressions with greedy and lazy quantifiers return the same matches. They most commonly return different results when they're used with the wildcard `(.)` metacharacter, which matches any character.

Quantifiers and Empty Matches

The quantifiers `*`, `+`, and `{n,m}` and their lazy counterparts never repeat after an empty match when the minimum number of captures has been found. This rule prevents quantifiers from entering infinite loops on empty subexpression matches when the maximum number of possible group captures is infinite or near infinite.

For example, the following code shows the result of a call to the `Regex.Match` method with the regular expression pattern `(a?)*`, which matches zero or one `a` character zero or more times. The single capturing group captures each `a` and `String.Empty`, but there's no second empty match because the first empty match causes the quantifier to stop repeating.

C#

```
using System;
using System.Text.RegularExpressions;

public class Example
{
    public static void Main()
    {
        string pattern = "(a?)*";
        string input = "aaabbb";
        Match match = Regex.Match(input, pattern);
        Console.WriteLine("Match: '{0}' at index {1}",
                           match.Value, match.Index);
        if (match.Groups.Count > 1) {
            GroupCollection groups = match.Groups;
            for (int grpCtr = 1; grpCtr <= groups.Count - 1; grpCtr++) {
                Console.WriteLine("    Group {0}: '{1}' at index {2}",
                                  grpCtr,
                                  groups[grpCtr].Value,
                                  groups[grpCtr].Index);
                int captureCtr = 0;
                foreach (Capture capture in groups[grpCtr].Captures) {
                    captureCtr++;
                    Console.WriteLine("        Capture {0}: '{1}' at index {2}",
                                      captureCtr, capture.Value, capture.Index);
                }
            }
        }
    }
}

// The example displays the following output:
//      Match: 'aaa' at index 0
//          Group 1: '' at index 3
//              Capture 1: 'a' at index 0
//              Capture 2: 'a' at index 1
//              Capture 3: 'a' at index 2
//              Capture 4: '' at index 3
```

To see the practical difference between a capturing group that defines a minimum and a maximum number of captures and one that defines a fixed number of captures, consider the regular expression patterns `(a\1|(?(1)\1)){0,2}` and `(a\1|(?(1)\1)){2}`. Both regular expressions consist of a single capturing group, which is defined in the following table:

| Pattern | Description |
|--------------------|--|
| <code>(a\1</code> | Either matches <code>a</code> along with the value of the first captured group ... |
| <code> (?(1</code> | ... or tests whether the first captured group has been defined. The <code>(?(1)</code> construct doesn't define a capturing group. |
| <code>\1))</code> | If the first captured group exists, match its value. If the group doesn't exist, the group will match String.Empty . |

The first regular expression tries to match this pattern between zero and two times; the second, exactly two times. Because the first pattern reaches its minimum number of captures with its first capture of [String.Empty](#), it never repeats to try to match `a\1`. The `{0,2}` quantifier allows only empty matches in the last iteration. In contrast, the second regular expression does match `a` because it evaluates `a\1` a second time. The minimum number of iterations, 2, forces the engine to repeat after an empty match.

C#

```
using System;
using System.Text.RegularExpressions;

public class Example
{
    public static void Main()
    {
        string pattern, input;

        pattern = @"(a\1|(?(1)\1)){0,2}";
        input = "aaabbb";

        Console.WriteLine("Regex pattern: {0}", pattern);
        Match match = Regex.Match(input, pattern);
        Console.WriteLine("Match: '{0}' at position {1}.",
                          match.Value, match.Index);
        if (match.Groups.Count > 1) {
            for (int groupCtr = 1; groupCtr <= match.Groups.Count - 1;
groupCtr++)
            {
                Group group = match.Groups[groupCtr];
                Console.WriteLine("  Group: {0}: '{1}' at position {2}.",
                                  groupCtr, group.Value, group.Index);
                int captureCtr = 0;
```

```

        foreach (Capture capture in group.Captures) {
            captureCtr++;
            Console.WriteLine("    Capture: {0}: '{1}' at position
{2}.",
                                captureCtr, capture.Value, capture.Index);
        }
    }
}
Console.WriteLine();

pattern = @"(a\1|(?(1)\1)){2}";
Console.WriteLine("Regex pattern: {0}", pattern);
match = Regex.Match(input, pattern);
Console.WriteLine("Matched '{0}' at position {1}.",
                    match.Value, match.Index);
if (match.Groups.Count > 1) {
    for (int groupCtr = 1; groupCtr <= match.Groups.Count - 1;
groupCtr++)
    {
        Group group = match.Groups[groupCtr];
        Console.WriteLine("    Group: {0}: '{1}' at position {2}.",
                            groupCtr, group.Value, group.Index);
        int captureCtr = 0;
        foreach (Capture capture in group.Captures) {
            captureCtr++;
            Console.WriteLine("        Capture: {0}: '{1}' at position
{2}.",
                                captureCtr, capture.Value, capture.Index);
        }
    }
}
}

// The example displays the following output:
//     Regex pattern: (a\1|(?(1)\1)){0,2}
//     Match: '' at position 0.
//         Group: 1: '' at position 0.
//             Capture: 1: '' at position 0.

//
//     Regex pattern: (a\1|(?(1)\1)){2}
//     Matched 'a' at position 0.
//         Group: 1: 'a' at position 0.
//             Capture: 1: '' at position 0.
//             Capture: 2: 'a' at position 0.

```

See also

- [Regular Expression Language - Quick Reference](#)
- [Backtracking](#)

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

Backreference Constructs in Regular Expressions

Article • 09/15/2021

Backreferences provide a convenient way to identify a repeated character or substring within a string. For example, if the input string contains multiple occurrences of an arbitrary substring, you can match the first occurrence with a capturing group, and then use a backreference to match subsequent occurrences of the substring.

ⓘ Note

A separate syntax is used to refer to named and numbered capturing groups in replacement strings. For more information, see [Substitutions](#).

.NET defines separate language elements to refer to numbered and named capturing groups. For more information about capturing groups, see [Grouping Constructs](#).

Numbered Backreferences

A numbered backreference uses the following syntax:

`\ number`

where *number* is the ordinal position of the capturing group in the regular expression. For example, `\4` matches the contents of the fourth capturing group. If *number* is not defined in the regular expression pattern, a parsing error occurs, and the regular expression engine throws an [ArgumentException](#). For example, the regular expression `\b(\w+)\s\1` is valid, because `(\w+)` is the first and only capturing group in the expression. On the other hand, `\b(\w+)\s\2` is invalid and throws an argument exception, because there is no capturing group numbered `\2`. In addition, if *number* identifies a capturing group in a particular ordinal position, but that capturing group has been assigned a numeric name different than its ordinal position, the regular expression parser also throws an [ArgumentException](#).

Note the ambiguity between octal escape codes (such as `\16`) and `\ number` backreferences that use the same notation. This ambiguity is resolved as follows:

- The expressions `\1` through `\9` are always interpreted as backreferences, and not as octal codes.

- If the first digit of a multidigit expression is 8 or 9 (such as `\80` or `\91`), the expression is interpreted as a literal.
- Expressions from `\10` and greater are considered backreferences if there is a backreference corresponding to that number; otherwise, they are interpreted as octal codes.
- If a regular expression contains a backreference to an undefined group number, a parsing error occurs, and the regular expression engine throws an [ArgumentException](#).

If the ambiguity is a problem, you can use the `\k<name>` notation, which is unambiguous and cannot be confused with octal character codes. Similarly, hexadecimal codes such as `\xdd` are unambiguous and cannot be confused with backreferences.

The following example finds doubled word characters in a string. It defines a regular expression, `(\w)\1`, which consists of the following elements.

| Element | Description |
|-------------------|--|
| <code>(\w)</code> | Match a word character and assign it to the first capturing group. |
| <code>\1</code> | Match the next character that is the same as the value of the first capturing group. |

C#

```
using System;
using System.Text.RegularExpressions;

public class Example
{
    public static void Main()
    {
        string pattern = @"(\w)\1";
        string input = "trellis llama webbing dresser swagger";
        foreach (Match match in Regex.Matches(input, pattern))
            Console.WriteLine("Found '{0}' at position {1}.",
                match.Value, match.Index);
    }
}
// The example displays the following output:
//      Found 'll' at position 3.
//      Found 'll' at position 8.
//      Found 'bb' at position 16.
//      Found 'ss' at position 25.
//      Found 'gg' at position 33.
```

Named Backreferences

A named backreference is defined by using the following syntax:

```
\k< name >
```

or:

```
\k' name '
```

where *name* is the name of a capturing group defined in the regular expression pattern. If *name* is not defined in the regular expression pattern, a parsing error occurs, and the regular expression engine throws an [ArgumentException](#).

The following example finds doubled word characters in a string. It defines a regular expression, `(?<char>\w)\k<char>`, which consists of the following elements.

| Element | Description |
|---|--|
| <code>(?</code> <code><char>\w)</code> | Match a word character and assign it to a capturing group named <code>char</code> . |
| <code>\k<char></code> | Match the next character that is the same as the value of the <code>char</code> capturing group. |

C#

```
using System;
using System.Text.RegularExpressions;

public class Example
{
    public static void Main()
    {
        string pattern = @"(?<char>\w)\k<char>";
        string input = "trellis llama webbing dresser swagger";
        foreach (Match match in Regex.Matches(input, pattern))
            Console.WriteLine("Found '{0}' at position {1}.",
                match.Value, match.Index);
    }
}
// The example displays the following output:
//      Found 'll' at position 3.
//      Found 'll' at position 8.
//      Found 'bb' at position 16.
//      Found 'ss' at position 25.
//      Found 'gg' at position 33.
```

Named numeric backreferences

In a named backreference with `\k<name>`, *name* can also be the string representation of a number. For example, the following example uses the regular expression `(?<2>\w)\k<2>` to find doubled word characters in a string. In this case, the example defines a capturing group that is explicitly named "2", and the backreference is correspondingly named "2".

```
C#  
  
using System;  
using System.Text.RegularExpressions;  
  
public class Example  
{  
    public static void Main()  
    {  
        string pattern = @"(?<2>\w)\k<2>";  
        string input = "trellis llama webbing dresser swagger";  
        foreach (Match match in Regex.Matches(input, pattern))  
            Console.WriteLine("Found '{0}' at position {1}.",  
                match.Value, match.Index);  
    }  
}  
// The example displays the following output:  
//     Found 'll' at position 3.  
//     Found 'll' at position 8.  
//     Found 'bb' at position 16.  
//     Found 'ss' at position 25.  
//     Found 'gg' at position 33.
```

If *name* is the string representation of a number, and no capturing group has that name, `\k<name>` is the same as the backreference `\number`, where *number* is the ordinal position of the capture. In the following example, there is a single capturing group named `char`. The backreference construct refers to it as `\k<1>`. As the output from the example shows, the call to the `Regex.IsMatch` succeeds because `char` is the first capturing group.

```
C#  
  
using System;  
using System.Text.RegularExpressions;  
  
public class Example  
{  
    public static void Main()  
    {  
        Console.WriteLine(Regex.IsMatch("aa", @"(?<char>\w)\k<1>"));  
        // Displays "True".  
    }  
}
```

```
}
```

However, if *name* is the string representation of a number and the capturing group in that position has been explicitly assigned a numeric name, the regular expression parser cannot identify the capturing group by its ordinal position. Instead, it throws an [ArgumentException](#). The only capturing group in the following example is named "2". Because the `\k` construct is used to define a backreference named "1", the regular expression parser is unable to identify the first capturing group and throws an exception.

```
C#
```

```
using System;
using System.Text.RegularExpressions;

public class Example
{
    public static void Main()
    {
        Console.WriteLine(Regex.IsMatch("aa", @"\(?<2>\w)\k<1>"));
        // Throws an ArgumentException.
    }
}
```

What Backreferences Match

A backreference refers to the most recent definition of a group (the definition most immediately to the left, when matching left to right). When a group makes multiple captures, a backreference refers to the most recent capture.

The following example includes a regular expression pattern, `(?<1>a)(?<1>\1b)*`, which redefines the `\1` named group. The following table describes each pattern in the regular expression.

| Pattern | Description |
|-------------------------------|--|
| <code>(?<1>a)</code> | Match the character "a" and assign the result to the capturing group named <code>1</code> . |
| <code>(?<1>\1b)*</code> | Match zero or more occurrences of the group named <code>1</code> along with a "b", and assign the result to the capturing group named <code>1</code> . |

```
C#
```

```

using System;
using System.Text.RegularExpressions;

public class Example
{
    public static void Main()
    {
        string pattern = @"(?<1>a)(?<1>\1b)*";
        string input = "aababb";
        foreach (Match match in Regex.Matches(input, pattern))
        {
            Console.WriteLine("Match: " + match.Value);
            foreach (Group group in match.Groups)
                Console.WriteLine("    Group: " + group.Value);
        }
    }
}

// The example displays the following output:
//      Group: aababb
//      Group: abb

```

In comparing the regular expression with the input string ("aababb"), the regular expression engine performs the following operations:

1. It starts at the beginning of the string, and successfully matches "a" with the expression `(?<1>a)`. The value of the `1` group is now "a".
2. It advances to the second character, and successfully matches the string "ab" with the expression `\1b`, or "ab". It then assigns the result, "ab" to `\1`.
3. It advances to the fourth character. The expression `(?<1>\1b)*` is to be matched zero or more times, so it successfully matches the string "abb" with the expression `\1b`. It assigns the result, "abb", back to `\1`.

In this example, `*` is a looping quantifier -- it is evaluated repeatedly until the regular expression engine cannot match the pattern it defines. Looping quantifiers do not clear group definitions.

If a group has not captured any substrings, a backreference to that group is undefined and never matches. This is illustrated by the regular expression pattern `\b(\p{Lu}{2})(\d{2})?(\p{Lu}{2})\b`, which is defined as follows:

| Pattern | Description |
|-----------------|-------------------------------------|
| <code>\b</code> | Begin the match on a word boundary. |

| Pattern | Description |
|-------------|---|
| (\p{Lu}{2}) | Match two uppercase letters. This is the first capturing group. |
| (\d{2})? | Match zero or one occurrence of two decimal digits. This is the second capturing group. |
| (\p{Lu}{2}) | Match two uppercase letters. This is the third capturing group. |
| \b | End the match on a word boundary. |

An input string can match this regular expression even if the two decimal digits that are defined by the second capturing group are not present. The following example shows that even though the match is successful, an empty capturing group is found between two successful capturing groups.

C#

```

using System;
using System.Text.RegularExpressions;

public class Example
{
    public static void Main()
    {
        string pattern = @"\b(\p{Lu}{2})(\d{2})?(\p{Lu}{2})\b";
        string[] inputs = { "AA22ZZ", "AABB" };
        foreach (string input in inputs)
        {
            Match match = Regex.Match(input, pattern);
            if (match.Success)
            {
                Console.WriteLine("Match in {0}: {1}", input, match.Value);
                if (match.Groups.Count > 1)
                {
                    for (int ctr = 1; ctr <= match.Groups.Count - 1; ctr++)
                    {
                        if (match.Groups[ctr].Success)
                            Console.WriteLine("Group {0}: {1}",
                                ctr, match.Groups[ctr].Value);
                        else
                            Console.WriteLine("Group {0}: <no match>", ctr);
                    }
                }
                Console.WriteLine();
            }
        }
    }
}

```

```
// The example displays the following output:  
//      Match in AA22ZZ: AA22ZZ  
//          Group 1: AA  
//          Group 2: 22  
//          Group 3: ZZ  
//  
//      Match in AABB: AABB  
//          Group 1: AA  
//          Group 2: <no match>  
//          Group 3: BB
```

See also

- [Regular Expression Language - Quick Reference](#)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

Alternation Constructs in Regular Expressions

Article • 01/14/2022

Alternation constructs modify a regular expression to enable either/or or conditional matching. .NET supports three alternation constructs:

- Pattern matching with |
- Conditional matching with (?(expression)yes|no)
- Conditional matching based on a valid captured group

Pattern Matching with |

You can use the vertical bar (|) character to match any one of a series of patterns, where the | character separates each pattern.

Like the positive character class, the | character can be used to match any one of a number of single characters. The following example uses both a positive character class and either/or pattern matching with the | character to locate occurrences of the words "gray" or "grey" in a string. In this case, the | character produces a regular expression that is more verbose.

C#

```
using System;
using System.Text.RegularExpressions;

public class Example
{
    public static void Main()
    {
        // Regular expression using character class.
        string pattern1 = @"\bgr[ae]y\b";
        // Regular expression using either/or.
        string pattern2 = @"\bgr(a|e)y\b";

        string input = "The gray wolf blended in among the grey rocks.";
        foreach (Match match in Regex.Matches(input, pattern1))
            Console.WriteLine("{0} found at position {1}",
                match.Value, match.Index);
        Console.WriteLine();
        foreach (Match match in Regex.Matches(input, pattern2))
            Console.WriteLine("{0} found at position {1}",
                match.Value, match.Index);
    }
}
```

```

}

// The example displays the following output:
//      'gray' found at position 4
//      'grey' found at position 35
//
//      'gray' found at position 4
//      'grey' found at position 35

```

The regular expression that uses the `|` character, `\bgr(a|e)y\b`, is interpreted as shown in the following table:

| Pattern | Description |
|--------------------|---------------------------------|
| <code>\b</code> | Start at a word boundary. |
| <code>gr</code> | Match the characters "gr". |
| <code>(a e)</code> | Match either an "a" or an "e". |
| <code>y\b</code> | Match a "y" on a word boundary. |

The `|` character can also be used to perform an either/or match with multiple characters or subexpressions, which can include any combination of character literals and regular expression language elements. (The character class does not provide this functionality.) The following example uses the `|` character to extract either a U.S. Social Security Number (SSN), which is a 9-digit number with the format *ddd-dd-dddd*, or a U.S. Employer Identification Number (EIN), which is a 9-digit number with the format *dd-ddddddd*.

C#

```

using System;
using System.Text.RegularExpressions;

public class Example
{
    public static void Main()
    {
        string pattern = @"\b(\d{2}-\d{7}|\d{3}-\d{2}-\d{4})\b";
        string input = "01-9999999 020-333333 777-88-9999";
        Console.WriteLine("Matches for {0}:", pattern);
        foreach (Match match in Regex.Matches(input, pattern))
            Console.WriteLine("  {0} at position {1}", match.Value,
match.Index);
    }
}

// The example displays the following output:
//      Matches for \b(\d{2}-\d{7}|\d{3}-\d{2}-\d{4})\b:

```

```
//          01-9999999 at position 0
//          777-88-9999 at position 22
```

The regular expression `\b(\d{2}-\d{7}|\d{3}-\d{2}-\d{4})\b` is interpreted as shown in the following table:

| Pattern | Description |
|--|--|
| <code>\b</code> | Start at a word boundary. |
| <code>(\d{2}-\d{7} \d{3}-\d{2}-\d{4})</code> | Match either of the following: two decimal digits followed by a hyphen followed by seven decimal digits; or three decimal digits, a hyphen, two decimal digits, another hyphen, and four decimal digits. |
| <code>\b</code> | End the match at a word boundary. |

Conditional matching with an expression

This language element attempts to match one of two patterns depending on whether it can match an initial pattern. Its syntax is:

```
(?(<code>expression</code>) yes )
```

or

```
(?(<code>expression</code>) yes | no )
```

where *expression* is the initial pattern to match, *yes* is the pattern to match if *expression* is matched, and *no* is the optional pattern to match if *expression* is not matched (if a *no* pattern is not provided, it's equivalent to an empty *no*). The regular expression engine treats *expression* as a zero-width assertion; that is, the regular expression engine does not advance in the input stream after it evaluates *expression*. Therefore, this construct is equivalent to the following:

```
(?(<code>?= expression</code>) yes | no )
```

where `(?= expression)` is a zero-width assertion construct. (For more information, see [Grouping Constructs](#).) Because the regular expression engine interprets *expression* as an anchor (a zero-width assertion), *expression* must either be a zero-width assertion (for more information, see [Anchors](#)) or a subexpression that is also contained in *yes*. Otherwise, the *yes* pattern cannot be matched.

 **Note**

If *expression* is a named or numbered capturing group, the alternation construct is interpreted as a capture test; for more information, see the next section, **Conditional Matching Based on a Valid Capture Group**. In other words, the regular expression engine does not attempt to match the captured substring, but instead tests for the presence or absence of the group.

The following example is a variation of the example that appears in the [Either/Or Pattern Matching with |](#) section. It uses conditional matching to determine whether the first three characters after a word boundary are two digits followed by a hyphen. If they are, it attempts to match a U.S. Employer Identification Number (EIN). If not, it attempts to match a U.S. Social Security Number (SSN).

C#

```
using System;
using System.Text.RegularExpressions;

public class Example
{
    public static void Main()
    {
        string pattern = @"\b(?:(\d{2}-)\d{2}-\d{7}|\d{3}-\d{2}-\d{4})\b";
        string input = "01-9999999 020-333333 777-88-9999";
        Console.WriteLine("Matches for {0}:", pattern);
        foreach (Match match in Regex.Matches(input, pattern))
            Console.WriteLine("    {0} at position {1}", match.Value,
match.Index);
    }
}
// The example displays the following output:
//      Matches for \b(?:(\d{2}-)\d{2}-\d{7}|\d{3}-\d{2}-\d{4})\b:
//          01-9999999 at position 0
//          777-88-9999 at position 22
```

The regular expression pattern `\b(?:(\d{2}-)\d{2}-\d{7}|\d{3}-\d{2}-\d{4})\b` is interpreted as shown in the following table:

| Pattern | Description |
|--------------------------|--|
| <code>\b</code> | Start at a word boundary. |
| <code>(?:(\d{2}-)</code> | Determine whether the next three characters consist of two digits followed by a hyphen. |
| <code>\d{2}-\d{7}</code> | If the previous pattern matches, match two digits followed by a hyphen followed by seven digits. |

| Pattern | Description |
|-------------------|--|
| \d{3}-\d{2}-\d{4} | If the previous pattern does not match, match three decimal digits, a hyphen, two decimal digits, another hyphen, and four decimal digits. |
| \b | Match a word boundary. |

Conditional matching based on a valid captured group

This language element attempts to match one of two patterns depending on whether it has matched a specified capturing group. Its syntax is:

(?(\ *name*) *yes*)

or

(?(\ *name*) *yes* | *no*)

or

(?(\ *number*) *yes*)

or

(?(\ *number*) *yes* | *no*)

where *name* is the name and *number* is the number of a capturing group, *yes* is the expression to match if *name* or *number* has a match, and *no* is the optional expression to match if it does not (if a *no* pattern is not provided, it's equivalent to an empty *no*).

If *name* does not correspond to the name of a capturing group that is used in the regular expression pattern, the alternation construct is interpreted as an expression test, as explained in the previous section. Typically, this means that *expression* evaluates to `false`. If *number* does not correspond to a numbered capturing group that is used in the regular expression pattern, the regular expression engine throws an [ArgumentException](#).

The following example is a variation of the example that appears in the [Either/Or Pattern Matching with |](#) section. It uses a capturing group named `n2` that consists of two digits followed by a hyphen. The alternation construct tests whether this capturing group has been matched in the input string. If it has, the alternation construct attempts to match the last seven digits of a nine-digit U.S. Employer Identification Number (EIN). If it has not, it attempts to match a nine-digit U.S. Social Security Number (SSN).

C#

```
using System;
using System.Text.RegularExpressions;

public class Example
{
    public static void Main()
    {
        string pattern = @"\b(?:<n2>\d{2}-)?(?:n2)\d{7}|\d{3}-\d{2}-\d{4})\b";
        string input = "01-9999999 020-333333 777-88-9999";
        Console.WriteLine("Matches for {0}:", pattern);
        foreach (Match match in Regex.Matches(input, pattern))
            Console.WriteLine("    {0} at position {1}", match.Value,
match.Index);
    }
}
// The example displays the following output:
//      Matches for \b(?:<n2>\d{2}-)?(?:n2)\d{7}|\d{3}-\d{2}-\d{4})\b:
//          01-9999999 at position 0
//          777-88-9999 at position 22
```

The regular expression pattern `\b(?:<n2>\d{2}-)?(?:n2)\d{7}|\d{3}-\d{2}-\d{4})\b` is interpreted as shown in the following table:

| Pattern | Description |
|------------------------------------|--|
| <code>\b</code> | Start at a word boundary. |
| <code>(?:<n2>\d{2}-)?</code> | Match zero or one occurrence of two digits followed by a hyphen. Name this capturing group <code>n2</code> . |
| <code>(?:n2)</code> | Test whether <code>n2</code> was matched in the input string. |
| <code>\d{7}</code> | If <code>n2</code> was matched, match seven decimal digits. |
| <code> \d{3}-\d{2}-\d{4}</code> | If <code>n2</code> was not matched, match three decimal digits, a hyphen, two decimal digits, another hyphen, and four decimal digits. |
| <code>\b</code> | Match a word boundary. |

A variation of this example that uses a numbered group instead of a named group is shown in the following example. Its regular expression pattern is `\b(\d{2}-)?(?:1)\d{7}|\d{3}-\d{2}-\d{4})\b`.

C#

```
using System;
using System.Text.RegularExpressions;
```

```
public class Example
{
    public static void Main()
    {
        string pattern = @"\b(\d{2}-)?(?:1)\d{7}|\d{3}-\d{2}-\d{4})\b";
        string input = "01-9999999 020-333333 777-88-9999";
        Console.WriteLine("Matches for {0}:", pattern);
        foreach (Match match in Regex.Matches(input, pattern))
            Console.WriteLine("  {0} at position {1}", match.Value,
match.Index);
    }
}
// The example display the following output:
//      Matches for \b(\d{2}-)?(?:1)\d{7}|\d{3}-\d{2}-\d{4})\b:
//      01-9999999 at position 0
//      777-88-9999 at position 22
```

See also

- Regular Expression Language - Quick Reference

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback [here](#).

 [Open a documentation issue](#)

 [Provide product feedback](#)

Substitutions in Regular Expressions

Article • 09/15/2021

Substitutions are language elements that are recognized only within replacement patterns. They use a regular expression pattern to define all or part of the text that is to replace matched text in the input string. The replacement pattern can consist of one or more substitutions along with literal characters. Replacement patterns are provided to overloads of the [Regex.Replace](#) method that have a `replacement` parameter and to the [Match.Result](#) method. The methods replace the matched pattern with the pattern that is defined by the `replacement` parameter.

.NET defines the substitution elements listed in the following table.

| Substitution | Description |
|--------------------------|---|
| <code>\$ number</code> | Includes the last substring matched by the capturing group that is identified by <i>number</i> , where <i>number</i> is a decimal value, in the replacement string. For more information, see Substituting a Numbered Group . |
| <code> \${ name }</code> | Includes the last substring matched by the named group that is designated by <code>(?<i>name</i>)</code> in the replacement string. For more information, see Substituting a Named Group . |
| <code> \$\$</code> | Includes a single "\$" literal in the replacement string. For more information, see Substituting a "\$" Symbol . |
| <code> \$&</code> | Includes a copy of the entire match in the replacement string. For more information, see Substituting the Entire Match . |
| <code> \$`</code> | Includes all the text of the input string before the match in the replacement string. For more information, see Substituting the Text before the Match . |
| <code> \$'</code> | Includes all the text of the input string after the match in the replacement string. For more information, see Substituting the Text after the Match . |
| <code> \$+</code> | Includes the last group captured in the replacement string. For more information, see Substituting the Last Captured Group . |
| <code> \$_</code> | Includes the entire input string in the replacement string. For more information, see Substituting the Entire Input String . |

Substitution Elements and Replacement Patterns

Substitutions are the only special constructs recognized in a replacement pattern. None of the other regular expression language elements, including character escapes and the period (.), which matches any character, are supported. Similarly, substitution language elements are recognized only in replacement patterns and are never valid in regular expression patterns.

The only character that can appear either in a regular expression pattern or in a substitution is the \$ character, although it has a different meaning in each context. In a regular expression pattern, \$ is an anchor that matches the end of the string. In a replacement pattern, \$ indicates the beginning of a substitution.

Note

For functionality similar to a replacement pattern within a regular expression, use a backreference. For more information about backreferences, see [Backreference Constructs](#).

Substituting a Numbered Group

The \$*number* language element includes the last substring matched by the *number* capturing group in the replacement string, where *number* is the index of the capturing group. For example, the replacement pattern \${1} indicates that the matched substring is to be replaced by the first captured group. For more information about numbered capturing groups, see [Grouping Constructs](#).

All digits that follow \$ are interpreted as belonging to the *number* group. If this is not your intent, you can substitute a named group instead. For example, you can use the replacement string \${1}1 instead of \${1}1 to define the replacement string as the value of the first captured group along with the number "1". For more information, see [Substituting a Named Group](#).

Capturing groups that are not explicitly assigned names using the (?<*name*>) syntax are numbered from left to right starting at one. Named groups are also numbered from left to right, starting at one greater than the index of the last unnamed group. For example, in the regular expression (\w)(?<digit>\d), the index of the digit named group is 2.

If *number* does not specify a valid capturing group defined in the regular expression pattern, \${*number*} is interpreted as a literal character sequence that is used to replace each match.

The following example uses the `$number` substitution to strip the currency symbol from a decimal value. It removes currency symbols found at the beginning or end of a monetary value, and recognizes the two most common decimal separators ("." and ",").

C#

```
using System;
using System.Text.RegularExpressions;

public class Example
{
    public static void Main()
    {
        string pattern = @"\p{Sc}*(\s?\d+[.,]?\d*)\p{Sc}*";
        string replacement = "$1";
        string input = "$16.32 12.19 €16.29 €18.29 €18,29";
        string result = Regex.Replace(input, pattern, replacement);
        Console.WriteLine(result);
    }
}
// The example displays the following output:
//      16.32 12.19 16.29 18.29  18,29
```

The regular expression pattern `\p{Sc}*(\s?\d+[.,]?\d*)\p{Sc}* is defined as shown in the following table.`

| Pattern | Description |
|------------------------------------|--|
| <code>\p{Sc}*</code> | Match zero or more currency symbol characters. |
| <code>\s?</code> | Match zero or one white-space characters. |
| <code>\d+</code> | Match one or more decimal digits. |
| <code>[.,]?</code> | Match zero or one period or comma. |
| <code>\d*</code> | Match zero or more decimal digits. |
| <code>(\s?\d+ [.,]?\d*)</code> | Match a white space followed by one or more decimal digits, followed by zero or one period or comma, followed by zero or more decimal digits. This is the first capturing group. Because the replacement pattern is <code>\$1</code> , the call to the <code>Regex.Replace</code> method replaces the entire matched substring with this captured group. |

Substituting a Named Group

The `${name}` language element substitutes the last substring matched by the `name` capturing group, where `name` is the name of a capturing group defined by the `(?)`

`<name>`) language element. For more information about named capturing groups, see [Grouping Constructs](#).

If *name* doesn't specify a valid named capturing group defined in the regular expression pattern but consists of digits, `${name}` is interpreted as a numbered group.

If *name* specifies neither a valid named capturing group nor a valid numbered capturing group defined in the regular expression pattern, `${name}` is interpreted as a literal character sequence that is used to replace each match.

The following example uses the `${name}` substitution to strip the currency symbol from a decimal value. It removes currency symbols found at the beginning or end of a monetary value, and recognizes the two most common decimal separators ("." and ",").

C#

```
using System;
using System.Text.RegularExpressions;

public class Example
{
    public static void Main()
    {
        string pattern = @"\p{Sc}*(?<amount>\s?\d+[.,]?\d*)\p{Sc}*";
        string replacement = "${amount}";
        string input = "$16.32 12.19 £16.29 €18.29 €18,29";
        string result = Regex.Replace(input, pattern, replacement);
        Console.WriteLine(result);
    }
}
// The example displays the following output:
//      16.32 12.19 16.29 18.29  18,29
```

The regular expression pattern `\p{Sc}*(?<amount>\s?\d[.,]?\d*)\p{Sc}*` is defined as shown in the following table.

| Pattern | Description |
|----------------------|--|
| <code>\p{Sc}*</code> | Match zero or more currency symbol characters. |
| <code>\s?</code> | Match zero or one white-space characters. |
| <code>\d+</code> | Match one or more decimal digits. |
| <code>[.,]?</code> | Match zero or one period or comma. |
| <code>\d*</code> | Match zero or more decimal digits. |

| Pattern | Description |
|------------------------------------|---|
| (? <amount>\s?\d[.,]?\d*)</amount> | Match a white space, followed by one or more decimal digits, followed by zero or one period or comma, followed by zero or more decimal digits. This is the capturing group named <code>amount</code> . Because the replacement pattern is <code> \${amount}</code> , the call to the Regex.Replace method replaces the entire matched substring with this captured group. |

Substituting a "\$" Character

The `$$` substitution inserts a literal "\$" character in the replaced string.

The following example uses the [NumberFormatInfo](#) object to determine the current culture's currency symbol and its placement in a currency string. It then builds both a regular expression pattern and a replacement pattern dynamically. If the example is run on a computer whose current culture is en-US, it generates the regular expression pattern `\b(\d+)(\.(\d+))?` and the replacement pattern `$$ $1$2`. The replacement pattern replaces the matched text with a currency symbol and a space followed by the first and second captured groups.

C#

```
using System;
using System.Globalization;
using System.Text.RegularExpressions;

public class Example
{
    public static void Main()
    {
        // Define array of decimal values.
        string[] values= { "16.35", "19.72", "1234", "0.99"};
        // Determine whether currency precedes (True) or follows (False)
        number.
        bool precedes = NumberFormatInfo.CurrentInfo.CurrencyPositivePattern %
        2 == 0;
        // Get decimal separator.
        string cSeparator =
        NumberFormatInfo.CurrentInfo.CurrencyDecimalSeparator;
        // Get currency symbol.
        string symbol = NumberFormatInfo.CurrentInfo.CurrencySymbol;
        // If symbol is a "$", add an extra "$".
        if (symbol == "$") symbol = " $$";

        // Define regular expression pattern and replacement string.
        string pattern = @"\b(\d+)" + cSeparator + @"(\d+)?";
        string replacement = "$1$2";
        replacement = precedes ? symbol + " " + replacement : replacement + "
```

```

    " + symbol;
    foreach (string value in values)
        Console.WriteLine("{0} --> {1}", value, Regex.Replace(value,
pattern, replacement));
    }
}
// The example displays the following output:
//      16.35 --> $ 16.35
//      19.72 --> $ 19.72
//      1234 --> $ 1234
//      0.99 --> $ 0.99

```

The regular expression pattern `\b(\d+)(\.(.\d+))?` is defined as shown in the following table.

| Pattern | Description |
|-----------------------|---|
| <code>\b</code> | Start the match at the beginning of a word boundary. |
| <code>(\d+)</code> | Match one or more decimal digits. This is the first capturing group. |
| <code>\.</code> | Match a period (the decimal separator). |
| <code>(\d+)</code> | Match one or more decimal digits. This is the third capturing group. |
| <code>(\.\d+)?</code> | Match zero or one occurrence of a period followed by one or more decimal digits. This is the second capturing group. |

Substituting the Entire Match

The `$&` substitution includes the entire match in the replacement string. Often, it is used to add a substring to the beginning or end of the matched string. For example, the `($&)` replacement pattern adds parentheses to the beginning and end of each match. If there is no match, the `$&` substitution has no effect.

The following example uses the `$&` substitution to add quotation marks at the beginning and end of book titles stored in a string array.

```

C#
using System;
using System.Text.RegularExpressions;

public class Example
{
    public static void Main()
    {
        string pattern = @"^(\w+\s?)+$";

```

```

        string[] titles = { "A Tale of Two Cities",
                            "The Hound of the Baskervilles",
                            "The Protestant Ethic and the Spirit of
                            Capitalism",
                            "The Origin of Species" };
        string replacement = "\"$&\"";
        foreach (string title in titles)
            Console.WriteLine(Regex.Replace(title, pattern, replacement));
    }
}

// The example displays the following output:
//      "A Tale of Two Cities"
//      "The Hound of the Baskervilles"
//      "The Protestant Ethic and the Spirit of Capitalism"
//      "The Origin of Species"

```

The regular expression pattern `^(\w+\s?)+$` is defined as shown in the following table.

| Pattern | Description |
|------------------------|--|
| <code>^</code> | Start the match at the beginning of the input string. |
| <code>(\w+\s?)+</code> | Match the pattern of one or more word characters followed by zero or one white-space characters one or more times. |
| <code>\$</code> | Match the end of the input string. |

The `"$&"` replacement pattern adds a literal quotation mark to the beginning and end of each match.

Substituting the Text Before the Match

The `$`` substitution replaces the matched string with the entire input string before the match. That is, it duplicates the input string up to the match while removing the matched text. Any text that follows the matched text is unchanged in the result string. If there are multiple matches in an input string, the replacement text is derived from the original input string, rather than from the string in which text has been replaced by earlier matches. (The example provides an illustration.) If there is no match, the `$`` substitution has no effect.

The following example uses the regular expression pattern `\d+` to match a sequence of one or more decimal digits in the input string. The replacement string `$`` replaces these digits with the text that precedes the match.

```

using System;
using System.Text.RegularExpressions;

public class Example
{
    public static void Main()
    {
        string input = "aa1bb2cc3dd4ee5";
        string pattern = @"\d+";
        string substitution = "$`";
        Console.WriteLine("Matches:");
        foreach (Match match in Regex.Matches(input, pattern))
            Console.WriteLine("    {0} at position {1}", match.Value,
match.Index);

        Console.WriteLine("Input string: {0}", input);
        Console.WriteLine("Output string: " +
                        Regex.Replace(input, pattern, substitution));
    }
}

// The example displays the following output:
//     Matches:
//         1 at position 2
//         2 at position 5
//         3 at position 8
//         4 at position 11
//         5 at position 14
//     Input string: aa1bb2cc3dd4ee5
//     Output string: aaaabbbaa1bbccaa1bb2ccddaa1bb2cc3ddeaa1bb2cc3dd4ee

```

In this example, the input string "aa1bb2cc3dd4ee5" contains five matches. The following table illustrates how the `$`` substitution causes the regular expression engine to replace each match in the input string. Inserted text is shown in bold in the results column.

| Match | Position | String before match | Result string |
|--------------|-----------------|--------------------------------|--|
| 1 | 2 | aa | aaaabb2cc3dd4ee5 |
| 2 | 5 | aa1bb | aaaabbbaa 1bbcc 3dd4ee5 |
| 3 | 8 | aa1bb2cc | aaaabbbaa1bbccaa 1bb2ccdd 4ee5 |
| 4 | 11 | aa1bb2cc3dd | aaaabbbaa1bbccaa1bb2ccddaa 1bb2cc3dde 5 |
| 5 | 14 | aa1bb2cc3dd4ee | aaaabbbaa1bbccaa1bb2ccddaa1bb2cc3ddeaa 1bb2cc3dd4ee |

Substituting the Text After the Match

The `$'` substitution replaces the matched string with the entire input string after the match. That is, it duplicates the input string after the match while removing the matched text. Any text that precedes the matched text is unchanged in the result string. If there is no match, the `$'` substitution has no effect.

The following example uses the regular expression pattern `\d+` to match a sequence of one or more decimal digits in the input string. The replacement string `$'` replaces these digits with the text that follows the match.

C#

```
using System;
using System.Text.RegularExpressions;

public class Example
{
    public static void Main()
    {
        string input = "aa1bb2cc3dd4ee5";
        string pattern = @"\d+";
        string substitution = "$'";
        Console.WriteLine("Matches:");
        foreach (Match match in Regex.Matches(input, pattern))
            Console.WriteLine("  {0} at position {1}", match.Value,
match.Index);
        Console.WriteLine("Input string: {0}", input);
        Console.WriteLine("Output string: " +
                           Regex.Replace(input, pattern, substitution));
    }
}
// The example displays the following output:
//   Matches:
//     1 at position 2
//     2 at position 5
//     3 at position 8
//     4 at position 11
//     5 at position 14
//   Input string: aa1bb2cc3dd4ee5
//   Output string: aabb2cc3dd4ee5bbcc3dd4ee5ccdd4ee5ddee5ee
```

In this example, the input string `"aa1bb2cc3dd4ee5"` contains five matches. The following table illustrates how the `$'` substitution causes the regular expression engine to replace each match in the input string. Inserted text is shown in bold in the results column.

| Match | Position | String after match | Result string |
|-------|----------|--------------------|------------------------------------|
| 1 | 2 | bb2cc3dd4ee5 | aabb2cc3dd4ee5bb2cc3dd4ee5 |
| 2 | 5 | cc3dd4ee5 | aabb2cc3dd4ee5bbcc3dd4ee5cc3dd4ee5 |

| Match | Position | String after match | Result string |
|-------|----------|--------------------|---|
| 3 | 8 | dd4ee5 | aabb2cc3dd4ee5bbcc3dd4ee5ccdd 4ee5dd4ee5 |
| 4 | 11 | ee5 | aabb2cc3dd4ee5bbcc3dd4ee5ccdd4ee5ddee 5ee5 |
| 5 | 14 | String.Empty | aabb2cc3dd4ee5bbcc3dd4ee5ccdd4ee5ddee 5ee5 |

Substituting the Last Captured Group

The `$+` substitution replaces the matched string with the last captured group. If there are no captured groups or if the value of the last captured group is `String.Empty`, the `$+` substitution has no effect.

The following example identifies duplicate words in a string and uses the `$+` substitution to replace them with a single occurrence of the word. The `RegexOptions.IgnoreCase` option is used to ensure that words that differ in case but that are otherwise identical are considered duplicates.

```
C#
using System;
using System.Text.RegularExpressions;

public class Example
{
    public static void Main()
    {
        string pattern = @"\b(\w+)\s\1\b";
        string substitution = "$+";
        string input = "The the dog jumped over the fence fence.";
        Console.WriteLine(Regex.Replace(input, pattern, substitution,
            RegexOptions.IgnoreCase));
    }
}
// The example displays the following output:
//      The dog jumped over the fence.
```

The regular expression pattern `\b(\w+)\s\1\b` is defined as shown in the following table.

| Pattern | Description |
|--------------------|---|
| <code>\b</code> | Begin the match at a word boundary. |
| <code>(\w+)</code> | Match one or more word characters. This is the first capturing group. |
| <code>\s</code> | Match a white-space character. |

| Pattern | Description |
|---------|-----------------------------------|
| \1 | Match the first captured group. |
| \b | End the match at a word boundary. |

Substituting the Entire Input String

The `$_` substitution replaces the matched string with the entire input string. That is, it removes the matched text and replaces it with the entire string, including the matched text.

The following example matches one or more decimal digits in the input string. It uses the `$_` substitution to replace them with the entire input string.

```
C#
using System;
using System.Text.RegularExpressions;

public class Example
{
    public static void Main()
    {
        string input = "ABC123DEF456";
        string pattern = @"\d+";
        string substitution = "$_";
        Console.WriteLine("Original string: {0}", input);
        Console.WriteLine("String with substitution: {0}",
                          Regex.Replace(input, pattern, substitution));
    }
}
// The example displays the following output:
//      Original string: ABC123DEF456
//      String with substitution: ABCABC123DEF456DEFABC123DEF456
```

In this example, the input string `"ABC123DEF456"` contains two matches. The following table illustrates how the `$_` substitution causes the regular expression engine to replace each match in the input string. Inserted text is shown in bold in the results column.

| Match | Position | Match | Result string |
|-------|----------|-------|--------------------------------|
| 1 | 3 | 123 | ABCABC123DEF456DEF456 |
| 2 | 5 | 456 | ABCABC123DEF456DEFABC123DEF456 |

See also

- Regular Expression Language - Quick Reference

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

Regular expression options

Article • 05/12/2023

By default, the comparison of an input string with any literal characters in a regular expression pattern is case-sensitive, white space in a regular expression pattern is interpreted as literal white-space characters, and capturing groups in a regular expression are named implicitly as well as explicitly. You can modify these and several other aspects of default regular expression behavior by specifying regular expression options. Some of these options, which are listed in the following table, can be included inline as part of the regular expression pattern, or they can be supplied to a [System.Text.RegularExpressions.Regex](#) class constructor or static pattern matching method as a [System.Text.RegularExpressions.RegexOptions](#) enumeration value.

| <code>RegexOptions</code> member | Inline character | Effect | More information |
|---|------------------|--|--|
| None | Not available | Use default behavior. | Default options |
| IgnoreCase | <code>i</code> | Use case-insensitive matching. | Case-insensitive matching |
| Multiline | <code>m</code> | Use multiline mode, where <code>^</code> and <code>\$</code> indicate the beginning and end of each line (instead of the beginning and end of the input string). | Multiline mode |
| Singleline | <code>s</code> | Use single-line mode, where the period <code>(.)</code> matches every character (instead of every character except <code>\n</code>). | Single-line mode |
| ExplicitCapture | <code>n</code> | Do not capture unnamed groups. The only valid captures are explicitly named or numbered groups of the form <code>(?<name> subexpression)</code> . | Explicit captures only |
| Compiled | Not available | Compile the regular expression to an assembly. | Compiled regular expressions |
| IgnorePatternWhitespace | <code>x</code> | Exclude unescaped white space from the pattern, and enable comments after a number sign <code>(#)</code> . | Ignore white space |

| RegexOptions member | Inline character | Effect | More information |
|----------------------------|-------------------------|---|--|
| RightToLeft | Not available | Change the search direction. Search moves from right to left instead of from left to right. | Right-to-left mode |
| ECMAScript | Not available | Enable ECMAScript-compliant behavior for the expression. | ECMAScript matching behavior |
| CultureInvariant | Not available | Ignore cultural differences in language. | Comparison using the invariant culture |
| NonBacktracking | Not available | Match using an approach that avoids backtracking and guarantees linear-time processing in the length of the input. (Available in .NET 7 and later versions.) | Nonbacktracking mode |

Specify options

You can specify options for regular expressions in one of three ways:

- In the `options` parameter of a [System.Text.RegularExpressions.Regex](#) class constructor or static (`Shared` in Visual Basic) pattern-matching method, such as `Regex(String, RegexOptions)` or `Regex.Match(String, String, RegexOptions)`. The `options` parameter is a bitwise OR combination of [System.Text.RegularExpressions.RegexOptions](#) enumerated values.

When options are supplied to a `Regex` instance by using the `options` parameter of a class constructor, the options are assigned to the `System.Text.RegularExpressions.RegexOptions` property. However, the `System.Text.RegularExpressions.RegexOptions` property does not reflect inline options in the regular expression pattern itself.

The following example provides an illustration. It uses the `options` parameter of the `Regex.Match(String, String, RegexOptions)` method to enable case-insensitive matching and to ignore pattern white space when identifying words that begin with the letter "d".

C#

```

string pattern = @"d \w+ \s";
string input = "Dogs are decidedly good pets.";
RegexOptions options = RegexOptions.IgnoreCase |
    RegexOptions.IgnorePatternWhitespace;

foreach (Match match in Regex.Matches(input, pattern, options))
    Console.WriteLine("{0}// found at index {1}.", match.Value,
        match.Index);
// The example displays the following output:
//      'Dogs // found at index 0.
//      'decidedly // found at index 9.

```

- By applying inline options in a regular expression pattern with the syntax `(?imnsx-imnsx)`. The option applies to the pattern from the point that the option is defined to either the end of the pattern or to the point at which the option is undefined by another inline option. Note that the [System.Text.RegularExpressions.RegexOptions](#) property of a [Regex](#) instance does not reflect these inline options. For more information, see the [Miscellaneous Constructs](#) topic.

The following example provides an illustration. It uses inline options to enable case-insensitive matching and to ignore pattern white space when identifying words that begin with the letter "d".

C#

```

string pattern = @"(?ix) d \w+ \s";
string input = "Dogs are decidedly good pets.";

foreach (Match match in Regex.Matches(input, pattern))
    Console.WriteLine("{0}// found at index {1}.", match.Value,
        match.Index);
// The example displays the following output:
//      'Dogs // found at index 0.
//      'decidedly // found at index 9.

```

- By applying inline options in a particular grouping construct in a regular expression pattern with the syntax `(?imnsx-imnsx: subexpression)`. No sign before a set of options turns the set on; a minus sign before a set of options turns the set off. (`(?` is a fixed part of the language construct's syntax that is required whether options are enabled or disabled.) The option applies only to that group. For more information, see [Grouping Constructs](#).

The following example provides an illustration. It uses inline options in a grouping construct to enable case-insensitive matching and to ignore pattern white space when identifying words that begin with the letter "d".

C#

```
string pattern = @"\b(?ix: d \w+)\s";
string input = "Dogs are decidedly good pets.";

foreach (Match match in Regex.Matches(input, pattern))
    Console.WriteLine("{0}// found at index {1}.", match.Value,
match.Index);
// The example displays the following output:
//      'Dogs // found at index 0.
//      'decidedly // found at index 9.
```

If options are specified inline, a minus sign (-) before an option or set of options turns off those options. For example, the inline construct (?ix-ms) turns on the [RegexOptions.IgnoreCase](#) and [RegexOptions.IgnorePatternWhitespace](#) options and turns off the [RegexOptions.Multiline](#) and [RegexOptions.Singleline](#) options. All regular expression options are turned off by default.

 **Note**

If the regular expression options specified in the `options` parameter of a constructor or method call conflict with the options specified inline in a regular expression pattern, the inline options are used.

The following five regular expression options can be set both with the `options` parameter and inline:

- [RegexOptions.IgnoreCase](#)
- [RegexOptions.Multiline](#)
- [RegexOptions.Singleline](#)
- [RegexOptions.ExplicitCapture](#)
- [RegexOptions.IgnorePatternWhitespace](#)

The following five regular expression options can be set using the `options` parameter but cannot be set inline:

- [RegexOptions.None](#)
- [RegexOptions.Compiled](#)
- [RegexOptions.RightToLeft](#)

- `RegexOptions.CultureInvariant`
- `RegexOptions.ECMAScript`

Determine options

You can determine which options were provided to a `Regex` object when it was instantiated by retrieving the value of the read-only `Regex.Options` property. This property is particularly useful for determining the options that are defined for a compiled regular expression created by the `Regex.CompileToAssembly` method.

To test for the presence of any option except `RegexOptions.None`, perform an AND operation with the value of the `Regex.Options` property and the `RegexOptions` value in which you are interested. Then test whether the result equals that `RegexOptions` value. The following example tests whether the `RegexOptions.IgnoreCase` option has been set.

C#

```
if ((rgx.Options & RegexOptions.IgnoreCase) == RegexOptions.IgnoreCase)
    Console.WriteLine("Case-insensitive pattern comparison.");
else
    Console.WriteLine("Case-sensitive pattern comparison.");
```

To test for `RegexOptions.None`, determine whether the value of the `Regex.Options` property is equal to `RegexOptions.None`, as the following example illustrates.

C#

```
if (rgx.Options == RegexOptions.None)
    Console.WriteLine("No options have been set.");
```

The following sections list the options supported by regular expression in .NET.

Default options

The `RegexOptions.None` option indicates that no options have been specified, and the regular expression engine uses its default behavior. This includes the following:

- The pattern is interpreted as a canonical rather than an ECMAScript regular expression.
- The regular expression pattern is matched in the input string from left to right.

- Comparisons are case-sensitive.
- The `^` and `$` language elements indicate the beginning and end of the input string. The end of the input string can be a trailing newline `\n` character.
- The `.` language element matches every character except `\n`.
- Any white space in a regular expression pattern is interpreted as a literal space character.
- The conventions of the current culture are used when comparing the pattern to the input string.
- Capturing groups in the regular expression pattern are implicit as well as explicit.

Note

The `RegexOptions.None` option has no inline equivalent. When regular expression options are applied inline, the default behavior is restored on an option-by-option basis, by turning a particular option off. For example, `(?i)` turns on case-insensitive comparison, and `(?-i)` restores the default case-sensitive comparison.

Because the `RegexOptions.None` option represents the default behavior of the regular expression engine, it is rarely explicitly specified in a method call. A constructor or static pattern-matching method without an `options` parameter is called instead.

Case-insensitive matching

The `IgnoreCase` option, or the `i` inline option, provides case-insensitive matching. By default, the casing conventions of the current culture are used.

The following example defines a regular expression pattern, `\bthe\w*\b`, that matches all words starting with "the". Because the first call to the `Match` method uses the default case-sensitive comparison, the output indicates that the string "The" that begins the sentence is not matched. It is matched when the `Match` method is called with options set to `IgnoreCase`.

C#

```
using System;
using System.Text.RegularExpressions;

public class Example
```

```

{
    public static void Main()
    {
        string pattern = @"\bthe\w*\b";
        string input = "The man then told them about that event.";
        foreach (Match match in Regex.Matches(input, pattern))
            Console.WriteLine("Found {0} at index {1}.", match.Value,
                match.Index);

        Console.WriteLine();
        foreach (Match match in Regex.Matches(input, pattern,
            RegexOptions.IgnoreCase))
            Console.WriteLine("Found {0} at index {1}.", match.Value,
                match.Index);
    }
}

// The example displays the following output:
//      Found then at index 8.
//      Found them at index 18.
//
//      Found The at index 0.
//      Found then at index 8.
//      Found them at index 18.

```

The following example modifies the regular expression pattern from the previous example to use inline options instead of the `options` parameter to provide case-insensitive comparison. The first pattern defines the case-insensitive option in a grouping construct that applies only to the letter "t" in the string "the". Because the option construct occurs at the beginning of the pattern, the second pattern applies the case-insensitive option to the entire regular expression.

C#

```

using System;
using System.Text.RegularExpressions;

public class CaseExample
{
    public static void Main()
    {
        string pattern = @"\b(?i:t)he\w*\b";
        string input = "The man then told them about that event.";
        foreach (Match match in Regex.Matches(input, pattern))
            Console.WriteLine("Found {0} at index {1}.", match.Value,
                match.Index);

        Console.WriteLine();
        pattern = @"(?i)\bthe\w*\b";
        foreach (Match match in Regex.Matches(input, pattern,
            RegexOptions.IgnoreCase))
            Console.WriteLine("Found {0} at index {1}.", match.Value,

```

```
match.Index);
    }
}

// The example displays the following output:
//      Found The at index 0.
//      Found then at index 8.
//      Found them at index 18.
//
//      Found The at index 0.
//      Found then at index 8.
//      Found them at index 18.
```

Multiline mode

The [RegexOptions.Multiline](#) option, or the `m` inline option, enables the regular expression engine to handle an input string that consists of multiple lines. It changes the interpretation of the `^` and `$` language elements so that they indicate the beginning and end of a line, instead of the beginning and end of the input string.

By default, `$` will be satisfied only at the end of the input string. If you specify the [RegexOptions.Multiline](#) option, it will be satisfied by either the newline character (`\n`) or the end of the input string.

In neither case does `$` recognize the carriage return/line feed character combination (`\r\n`). `$` always ignores any carriage return (`\r`). To end your match with either `\r\n` or `\n`, use the subexpression `\r?\$` instead of just `$`. Note that this will make the `\r` part of the match.

The following example extracts bowlers' names and scores and adds them to a [SortedList<TKey,TValue>](#) collection that sorts them in descending order. The [Matches](#) method is called twice. In the first method call, the regular expression is `^(\\w+)\\s(\\d+)$` and no options are set. As the output shows, because the regular expression engine cannot match the input pattern along with the beginning and end of the input string, no matches are found. In the second method call, the regular expression is changed to `^(\\w+)\\s(\\d+)\\r?$` and the options are set to [RegexOptions.Multiline](#). As the output shows, the names and scores are successfully matched, and the scores are displayed in descending order.

C#

```
using System;
using System.Collections.Generic;
using System.Text.RegularExpressions;
```

```

public class Multiline1Example
{
    public static void Main()
    {
        SortedList<int, string> scores = new SortedList<int, string>(new
DescendingComparer1<int>());

        string input = "Joe 164\n" +
                      "Sam 208\n" +
                      "Allison 211\n" +
                      "Gwen 171\n";
        string pattern = @"\^(\w+)\s(\d+)$";
        bool matched = false;

        Console.WriteLine("Without Multiline option:");
        foreach (Match match in Regex.Matches(input, pattern))
        {
            scores.Add(Int32.Parse(match.Groups[2].Value),
(string)match.Groups[1].Value);
            matched = true;
        }
        if (!matched)
            Console.WriteLine("    No matches.");
        Console.WriteLine();

        // Redefine pattern to handle multiple lines.
        pattern = @"\^(\w+)\s(\d+)\r*$";
        Console.WriteLine("With multiline option:");
        foreach (Match match in Regex.Matches(input, pattern,
RegexOptions.Multiline))
            scores.Add(Int32.Parse(match.Groups[2].Value),
(string)match.Groups[1].Value);

        // List scores in descending order.
        foreach (KeyValuePair<int, string> score in scores)
            Console.WriteLine("{0}: {1}", score.Value, score.Key);
    }
}

public class DescendingComparer1<T> : IComparer<T>
{
    public int Compare(T x, T y)
    {
        return Comparer<T>.Default.Compare(x, y) * -1;
    }
}
// The example displays the following output:
// Without Multiline option:
//     No matches.
//
// With multiline option:
// Allison: 211
// Sam: 208
// Gwen: 171
// Joe: 164

```

The regular expression pattern `^(\w+)\s(\d+)\r*$` is defined as shown in the following table.

| Pattern | Description |
|--------------------|---|
| <code>^</code> | Begin at the start of the line. |
| <code>(\w+)</code> | Match one or more word characters. This is the first capturing group. |
| <code>\s</code> | Match a white-space character. |
| <code>(\d+)</code> | Match one or more decimal digits. This is the second capturing group. |
| <code>\r?</code> | Match zero or one carriage return character. |
| <code>\$</code> | End at the end of the line. |

The following example is equivalent to the previous one, except that it uses the inline option `(?m)` to set the multiline option.

C#

```
using System;
using System.Collections.Generic;
using System.Text.RegularExpressions;

public class Multiline2Example
{
    public static void Main()
    {
        SortedList<int, string> scores = new SortedList<int, string>(new
DescendingComparer<int>());

        string input = "Joe 164\n" +
                      "Sam 208\n" +
                      "Allison 211\n" +
                      "Gwen 171\n";
        string pattern = @"(?m)^(\w+)\s(\d+)\r*$";

        foreach (Match match in Regex.Matches(input, pattern,
RegexOptions.Multiline))
            scores.Add(Convert.ToInt32(match.Groups[2].Value),
match.Groups[1].Value);

        // List scores in descending order.
        foreach (KeyValuePair<int, string> score in scores)
            Console.WriteLine("{0}: {1}", score.Value, score.Key);
    }
}
```

```
public class DescendingComparer<T> : IComparer<T>
{
    public int Compare(T x, T y)
    {
        return Comparer<T>.Default.Compare(x, y) * -1;
    }
}
// The example displays the following output:
//    Allison: 211
//    Sam: 208
//    Gwen: 171
//    Joe: 164
```

Single-line mode

The [RegexOptions.Singleline](#) option, or the `s` inline option, causes the regular expression engine to treat the input string as if it consists of a single line. It does this by changing the behavior of the period (`.`) language element so that it matches every character, instead of matching every character except for the newline character `\n`.

The following example illustrates how the behavior of the `.` language element changes when you use the [RegexOptions.Singleline](#) option. The regular expression `^.+` starts at the beginning of the string and matches every character. By default, the match ends at the end of the first line; the regular expression pattern matches the carriage return character `\r`, but it does not match `\n`. Because the [RegexOptions.Singleline](#) option interprets the entire input string as a single line, it matches every character in the input string, including `\n`.

C#

```
using System;
using System.Text.RegularExpressions;

public class Example
{
    public static void Main()
    {
        string pattern = "^.+";
        string input = "This is one line and" + Environment.NewLine + "this is
the second.";
        foreach (Match match in Regex.Matches(input, pattern))
            Console.WriteLine(Regex.Escape(match.Value));

        Console.WriteLine();
        foreach (Match match in Regex.Matches(input, pattern,
RegexOptions.Singleline))
            Console.WriteLine(Regex.Escape(match.Value));
```

```
    }
}

// The example displays the following output:
//      This\ is\ one\ line\ and\r
//
//      This\ is\ one\ line\ and\r\nthis\ is\ the\ second\.
```

The following example is equivalent to the previous one, except that it uses the inline option `(?s)` to enable single-line mode.

```
C#

using System;
using System.Text.RegularExpressions;

public class SingleLineExample
{
    public static void Main()
    {
        string pattern = "(?s)^.+";
        string input = "This is one line and" + Environment.NewLine + "this
is the second.";

        foreach (Match match in Regex.Matches(input, pattern))
            Console.WriteLine(Regex.Escape(match.Value));
    }
}
// The example displays the following output:
//      This\ is\ one\ line\ and\r\nthis\ is\ the\ second\.
```

Explicit captures only

By default, capturing groups are defined by the use of parentheses in the regular expression pattern. Named groups are assigned a name or number by the `(?subexpression)` language option, whereas unnamed groups are accessible by index. In the [GroupCollection](#) object, unnamed groups precede named groups.

Grouping constructs are often used only to apply quantifiers to multiple language elements, and the captured substrings are of no interest. For example, if the following regular expression:

```
\b\((?((\w+),?\s?)+[\.\!?\])\)?
```

is intended only to extract sentences that end with a period, exclamation point, or question mark from a document, only the resulting sentence (which is represented by the [Match](#) object) is of interest. The individual words in the collection are not.

Capturing groups that are not subsequently used can be expensive, because the regular expression engine must populate both the [GroupCollection](#) and [CaptureCollection](#) collection objects. As an alternative, you can use either the [RegexOptions.ExplicitCapture](#) option or the `n` inline option to specify that the only valid captures are explicitly named or numbered groups that are designated by the `(?<name> subexpression)` construct.

The following example displays information about the matches returned by the `\b\((?((\w+),?\s?)+[\.\!?])\)?` regular expression pattern when the [Match](#) method is called with and without the [RegexOptions.ExplicitCapture](#) option. As the output from the first method call shows, the regular expression engine fully populates the [GroupCollection](#) and [CaptureCollection](#) collection objects with information about captured substrings. Because the second method is called with `options` set to [RegexOptions.ExplicitCapture](#), it does not capture information on groups.

C#

```
using System;
using System.Text.RegularExpressions;

public class Explicit1Example
{
    public static void Main()
    {
        string input = "This is the first sentence. Is it the beginning " +
                      "of a literary masterpiece? I think not. Instead, " +
                      "it is a nonsensical paragraph.";
        string pattern = @"\b\((?((\w+),?\s?)+[\.\!?])\)?";
        Console.WriteLine("With implicit captures:");
        foreach (Match match in Regex.Matches(input, pattern))
        {
            Console.WriteLine("The match: {0}", match.Value);
            int groupCtr = 0;
            foreach (Group group in match.Groups)
            {
                Console.WriteLine("    Group {0}: {1}", groupCtr,
group.Value);
                groupCtr++;
                int captureCtr = 0;
                foreach (Capture capture in group.Captures)
                {
                    Console.WriteLine("        Capture {0}: {1}", captureCtr,
capture.Value);
                    captureCtr++;
                }
            }
        }
        Console.WriteLine();
        Console.WriteLine("With explicit captures only:");
        foreach (Match match in Regex.Matches(input, pattern,
RegexOptions.ExplicitCapture))
```

```
    {
        Console.WriteLine("The match: {0}", match.Value);
        int groupCtr = 0;
        foreach (Group group in match.Groups)
        {
            Console.WriteLine("    Group {0}: {1}", groupCtr,
group.Value);
            groupCtr++;
            int captureCtr = 0;
            foreach (Capture capture in group.Captures)
            {
                Console.WriteLine("        Capture {0}: {1}", captureCtr,
capture.Value);
                captureCtr++;
            }
        }
    }
}

// The example displays the following output:
//   With implicit captures:
//     The match: This is the first sentence.
//       Group 0: This is the first sentence.
//         Capture 0: This is the first sentence.
//       Group 1: sentence
//         Capture 0: This
//         Capture 1: is
//         Capture 2: the
//         Capture 3: first
//         Capture 4: sentence
//       Group 2: sentence
//         Capture 0: This
//         Capture 1: is
//         Capture 2: the
//         Capture 3: first
//         Capture 4: sentence
//     The match: Is it the beginning of a literary masterpiece?
//       Group 0: Is it the beginning of a literary masterpiece?
//         Capture 0: Is it the beginning of a literary masterpiece?
//       Group 1: masterpiece
//         Capture 0: Is
//         Capture 1: it
//         Capture 2: the
//         Capture 3: beginning
//         Capture 4: of
//         Capture 5: a
//         Capture 6: literary
//         Capture 7: masterpiece
//       Group 2: masterpiece
//         Capture 0: Is
//         Capture 1: it
//         Capture 2: the
//         Capture 3: beginning
//         Capture 4: of
//         Capture 5: a
```

```

//           Capture 6: literary
//           Capture 7: masterpiece
// The match: I think not.
//           Group 0: I think not.
//           Capture 0: I think not.
//           Group 1: not
//           Capture 0: I
//           Capture 1: think
//           Capture 2: not
//           Group 2: not
//           Capture 0: I
//           Capture 1: think
//           Capture 2: not
// The match: Instead, it is a nonsensical paragraph.
//           Group 0: Instead, it is a nonsensical paragraph.
//           Capture 0: Instead, it is a nonsensical paragraph.
//           Group 1: paragraph
//           Capture 0: Instead,
//           Capture 1: it
//           Capture 2: is
//           Capture 3: a
//           Capture 4: nonsensical
//           Capture 5: paragraph
//           Group 2: paragraph
//           Capture 0: Instead
//           Capture 1: it
//           Capture 2: is
//           Capture 3: a
//           Capture 4: nonsensical
//           Capture 5: paragraph
//
// With explicit captures only:
// The match: This is the first sentence.
//           Group 0: This is the first sentence.
//           Capture 0: This is the first sentence.
// The match: Is it the beginning of a literary masterpiece?
//           Group 0: Is it the beginning of a literary masterpiece?
//           Capture 0: Is it the beginning of a literary masterpiece?
// The match: I think not.
//           Group 0: I think not.
//           Capture 0: I think not.
// The match: Instead, it is a nonsensical paragraph.
//           Group 0: Instead, it is a nonsensical paragraph.
//           Capture 0: Instead, it is a nonsensical paragraph.

```

The regular expression pattern `\b\((?((?>\w+),?\s?)*)+[\.\!?\]\)\?` is defined as shown in the following table.

| Pattern | Description |
|-----------------|---|
| <code>\b</code> | Begin at a word boundary. |
| <code>\(</code> | Match zero or one occurrences of the opening parenthesis ("("). |

| Pattern | Description |
|---------------|---|
| (?>\w+),? | Match one or more word characters, followed by zero or one commas. Do not backtrack when matching word characters. |
| \s? | Match zero or one white-space characters. |
| ((\w+),?\s?)+ | Match the combination of one or more word characters, zero or one commas, and zero or one white-space characters one or more times. |
| [.\,!?]\\\)? | Match any of the three punctuation symbols, followed by zero or one closing parentheses (""). |

You can also use the `(?n)` inline element to suppress automatic captures. The following example modifies the previous regular expression pattern to use the `(?n)` inline element instead of the [RegexOptions.ExplicitCapture](#) option.

C#

```
using System;
using System.Text.RegularExpressions;

public class Explicit2Example
{
    public static void Main()
    {
        string input = "This is the first sentence. Is it the beginning " +
                      "of a literary masterpiece? I think not. Instead, " +
                      "it is a nonsensical paragraph.";
        string pattern = @"(?n)\b\((?(>\w+),?\s?)+[\.\,!?]\\\)?";

        foreach (Match match in Regex.Matches(input, pattern))
        {
            Console.WriteLine("The match: {0}", match.Value);
            int groupCtr = 0;
            foreach (Group group in match.Groups)
            {
                Console.WriteLine("    Group {0}: {1}", groupCtr,
group.Value);
                groupCtr++;
                int captureCtr = 0;
                foreach (Capture capture in group.Captures)
                {
                    Console.WriteLine("        Capture {0}: {1}", captureCtr,
capture.Value);
                    captureCtr++;
                }
            }
        }
    }
}

// The example displays the following output:
//      The match: This is the first sentence.
```

```
//      Group 0: This is the first sentence.
//      Capture 0: This is the first sentence.
//      The match: Is it the beginning of a literary masterpiece?
//      Group 0: Is it the beginning of a literary masterpiece?
//      Capture 0: Is it the beginning of a literary masterpiece?
//      The match: I think not.
//      Group 0: I think not.
//      Capture 0: I think not.
//      The match: Instead, it is a nonsensical paragraph.
//      Group 0: Instead, it is a nonsensical paragraph.
//      Capture 0: Instead, it is a nonsensical paragraph.
```

Finally, you can use the inline group element `(?n:)` to suppress automatic captures on a group-by-group basis. The following example modifies the previous pattern to suppress unnamed captures in the outer group, `((?>\w+),?\s?)`. Note that this suppresses unnamed captures in the inner group as well.

C#

```
using System;
using System.Text.RegularExpressions;

public class Explicit3Example
{
    public static void Main()
    {
        string input = "This is the first sentence. Is it the beginning " +
                      "of a literary masterpiece? I think not. Instead, " +
                      "it is a nonsensical paragraph.";
        string pattern = @"\b\((?:(?n:(?>\w+),?\s?)+[\.\!?\])\)?";

        foreach (Match match in Regex.Matches(input, pattern))
        {
            Console.WriteLine("The match: {0}", match.Value);
            int groupCtr = 0;
            foreach (Group group in match.Groups)
            {
                Console.WriteLine("  Group {0}: {1}", groupCtr,
group.Value);
                groupCtr++;
                int captureCtr = 0;
                foreach (Capture capture in group.Captures)
                {
                    Console.WriteLine("    Capture {0}: {1}", captureCtr,
capture.Value);
                    captureCtr++;
                }
            }
        }
    }
} // The example displays the following output:
```

```
//      The match: This is the first sentence.
//          Group 0: This is the first sentence.
//              Capture 0: This is the first sentence.
//      The match: Is it the beginning of a literary masterpiece?
//          Group 0: Is it the beginning of a literary masterpiece?
//              Capture 0: Is it the beginning of a literary masterpiece?
//      The match: I think not.
//          Group 0: I think not.
//              Capture 0: I think not.
//      The match: Instead, it is a nonsensical paragraph.
//          Group 0: Instead, it is a nonsensical paragraph.
//              Capture 0: Instead, it is a nonsensical paragraph.
```

Compiled regular expressions

Note

Where possible, use [source generated regular expressions](#) instead of compiling regular expressions using the [RegexOptions.Compiled](#) option. Source generation can help your app start faster, run more quickly and be more trimmable. To learn when source generation is possible, see [When to use it](#).

By default, regular expressions in .NET are interpreted. When a [Regex](#) object is instantiated or a static [Regex](#) method is called, the regular expression pattern is parsed into a set of custom opcodes, and an interpreter uses these opcodes to run the regular expression. This involves a tradeoff: The cost of initializing the regular expression engine is minimized at the expense of run-time performance.

You can use compiled instead of interpreted regular expressions by using the [RegexOptions.Compiled](#) option. In this case, when a pattern is passed to the regular expression engine, it is parsed into a set of opcodes and then converted to Microsoft intermediate language (MSIL), which can be passed directly to the common language runtime. Compiled regular expressions maximize run-time performance at the expense of initialization time.

Note

A regular expression can be compiled only by supplying the [RegexOptions.Compiled](#) value to the `options` parameter of a [Regex](#) class constructor or a static pattern-matching method. It is not available as an inline option.

You can use compiled regular expressions in calls to both static and instance regular expressions. In static regular expressions, the [RegexOptions.Compiled](#) option is passed to the `options` parameter of the regular expression pattern-matching method. In instance regular expressions, it is passed to the `options` parameter of the [Regex](#) class constructor. In both cases, it results in enhanced performance.

However, this improvement in performance occurs only under the following conditions:

- A [Regex](#) object that represents a particular regular expression is used in multiple calls to regular expression pattern-matching methods.
- The [Regex](#) object is not allowed to go out of scope, so it can be reused.
- A static regular expression is used in multiple calls to regular expression pattern-matching methods. (The performance improvement is possible because regular expressions used in static method calls are cached by the regular expression engine.)

 **Note**

The [RegexOptions.Compiled](#) option is unrelated to the [Regex.CompileToAssembly](#) method, which creates a special-purpose assembly that contains predefined compiled regular expressions.

Ignore white space

By default, white space in a regular expression pattern is significant; it forces the regular expression engine to match a white-space character in the input string. Because of this, the regular expression "`\b\w+\s`" and "`\b\w+`" are roughly equivalent regular expressions. In addition, when the number sign (#) is encountered in a regular expression pattern, it is interpreted as a literal character to be matched.

The [RegexOptions.IgnorePatternWhitespace](#) option, or the `x` inline option, changes this default behavior as follows:

- Unescaped white space in the regular expression pattern is ignored. To be part of a regular expression pattern, white-space characters must be escaped (for example, as `\s` or "`\` ").
- The number sign (#) is interpreted as the beginning of a comment, rather than as a literal character. All text in the regular expression pattern from the `#` character to

either the next `\n` character or to the end of the string, is interpreted as a comment.

However, in the following cases, white-space characters in a regular expression aren't ignored, even if you use the [RegexOptions.IgnorePatternWhitespace](#) option:

- White space within a character class is always interpreted literally. For example, the regular expression pattern `[.,;:]` matches any single white-space character, period, comma, semicolon, or colon.
- White space isn't allowed within a bracketed quantifier, such as `{ n }`, `{ n , }`, and `{ n , m }`. For example, the regular expression pattern `\d{1, 3}` fails to match any sequences of digits from one to three digits because it contains a white-space character.
- White space isn't allowed within a character sequence that introduces a language element. For example:
 - The language element `(?: subexpression)` represents a noncapturing group, and the `(?:` portion of the element can't have embedded spaces. The pattern `(? : subexpression)` throws an [ArgumentException](#) at run time because the regular expression engine can't parse the pattern, and the pattern `(? : subexpression)` fails to match *subexpression*.
 - The language element `\p{ name }`, which represents a Unicode category or named block, can't include embedded spaces in the `\p{` portion of the element. If you do include a white space, the element throws an [ArgumentException](#) at run time.

Enabling this option helps simplify regular expressions that are often difficult to parse and to understand. It improves readability, and makes it possible to document a regular expression.

The following example defines the following regular expression pattern:

```
\b \((? ( (?>\w+) ,?\s? )+ [\.\!?] \)? # Matches an entire sentence.
```

This pattern is similar to the pattern defined in the [Explicit Captures Only](#) section, except that it uses the [RegexOptions.IgnorePatternWhitespace](#) option to ignore pattern white space.

C#

```

using System;
using System.Text.RegularExpressions;

public class Whitespace1Example
{
    public static void Main()
    {
        string input = "This is the first sentence. Is it the beginning " +
                      "of a literary masterpiece? I think not. Instead, " +
                      "it is a nonsensical paragraph.";
        string pattern = @"\b \(? ( (?>\w+),?\s? )+ [\.\!?\] \)? # Matches an
entire sentence./";

        foreach (Match match in Regex.Matches(input, pattern,
RegexOptions.IgnorePatternWhitespace))
            Console.WriteLine(match.Value);
    }
}

// The example displays the following output:
//      This is the first sentence.
//      Is it the beginning of a literary masterpiece?
//      I think not.
//      Instead, it is a nonsensical paragraph.

```

The following example uses the inline option `(?x)` to ignore pattern white space.

C#

```

using System;
using System.Text.RegularExpressions;

public class Whitespace2Example
{
    public static void Main()
    {
        string input = "This is the first sentence. Is it the beginning " +
                      "of a literary masterpiece? I think not. Instead, " +
                      "it is a nonsensical paragraph.";
        string pattern = @"(?x)\b \(? ( (?>\w+),?\s? )+ [\.\!?\] \)? # 
Matches an entire sentence./";

        foreach (Match match in Regex.Matches(input, pattern))
            Console.WriteLine(match.Value);
    }
}

// The example displays the following output:
//      This is the first sentence.
//      Is it the beginning of a literary masterpiece?
//      I think not.
//      Instead, it is a nonsensical paragraph.

```

Right-to-left mode

By default, the regular expression engine searches from left to right. You can reverse the search direction by using the [RegexOptions.RightToLeft](#) option. The right-to-left search automatically begins at the last character position of the string. For pattern-matching methods that include a starting position parameter, such as [Regex.Match\(String, Int32\)](#), the specified starting position is the index of the rightmost character position at which the search is to begin.

ⓘ Note

Right-to-left pattern mode is available only by supplying the [RegexOptions.RightToLeft](#) value to the `options` parameter of a [Regex](#) class constructor or static pattern-matching method. It is not available as an inline option.

Example

The regular expression `\bb\w+\s` matches words with two or more characters that begin with the letter "b" and are followed by a white-space character. In the following example, the input string consists of three words that include one or more "b" characters. The first and second words begin with "b" and the third word ends with "b". As the output from the right-to-left search example shows, only the first and second words match the regular expression pattern, with the second word being matched first.

C#

```
using System;
using System.Text.RegularExpressions;

public class RTL1Example
{
    public static void Main()
    {
        string pattern = @"\bb\w+\s";
        string input = "build band tab";
        foreach (Match match in Regex.Matches(input, pattern,
RegexOptions.RightToLeft))
            Console.WriteLine("{0} found at position {1}.",
match.Value,
match.Index);
    }
}
// The example displays the following output:
```

```
//      'band ' found at position 6.  
//      'build ' found at position 0.
```

Evaluation order

The [RegexOptions.RightToLeft](#) option changes the search direction and also reverses the order in which the regular expression pattern is evaluated. In a right-to-left search, the search pattern is read from right to left. This distinction is important because it can affect things like capture groups and [backreferences](#). For example, the expression `Regex.Match("abcabc", @"\1(abc)", RegexOptions.RightToLeft)` finds a match `abcabc`, but in a left-to-right search (`Regex.Match("abcabc", @"\1(abc)", RegexOptions.None)`), no match is found. That's because the `(abc)` element must be evaluated before the numbered capturing group element `(\1)` for a match to be found.

Lookahead and lookbehind assertions

The location of a match for a lookahead `((?=subexpression))` or lookbehind `((?<=subexpression))` assertion doesn't change in a right-to-left search. The lookahead assertions look to the right of the current match location; the lookbehind assertions look to the left of the current match location.

Tip

Whether a search is right-to-left or not, lookbehinds are implemented using a right-to-left search starting at the current match location.

For example, the regular expression `(?<=\d{1,2}\s)\w+, \s\d{4}` uses the lookbehind assertion to test for a date that precedes a month name. The regular expression then matches the month and the year. For information on lookahead and lookbehind assertions, see [Grouping Constructs](#).

C#

```
using System;  
using System.Text.RegularExpressions;  
  
public class RTL2Example  
{  
    public static void Main()  
    {  
        string[] inputs = { "1 May, 1917", "June 16, 2003" };  
        string pattern = @"(?<=\d{1,2}\s)\w+, \s\d{4}";
```

```

foreach (string input in inputs)
{
    Match match = Regex.Match(input, pattern,
    RegexOptions.RightToLeft);
    if (match.Success)
        Console.WriteLine("The date occurs in {0}.", match.Value);
    else
        Console.WriteLine("{0} does not match.", input);
}
}

// The example displays the following output:
//      The date occurs in May, 1917.
//      June 16, 2003 does not match.

```

The regular expression pattern is defined as shown in the following table.

| Pattern | Description |
|--------------------|---|
| (? =<\d{1,2}\s) | The beginning of the match must be preceded by one or two decimal digits followed by a space. |
| \w+ | Match one or more word characters. |
| , | Match one comma character. |
| \s | Match a white-space character. |
| \d{4} | Match four decimal digits. |

ECMAScript matching behavior

By default, the regular expression engine uses canonical behavior when matching a regular expression pattern to input text. However, you can instruct the regular expression engine to use ECMAScript matching behavior by specifying the [RegexOptions.ESCAPE](#) option.

ⓘ Note

ECMAScript-compliant behavior is available only by supplying the [RegexOptions.ESCAPE](#) value to the `options` parameter of a [Regex](#) class constructor or static pattern-matching method. It is not available as an inline option.

The `RegexOptions.ESCAPE` option can be combined only with the `RegexOptions.IgnoreCase` and `RegexOptions.Multiline` options. The use of any other option in a regular expression results in an `ArgumentOutOfRangeException`.

The behavior of ECMAScript and canonical regular expressions differs in three areas: character class syntax, self-referencing capturing groups, and octal versus backreference interpretation.

- Character class syntax. Because canonical regular expressions support Unicode whereas ECMAScript does not, character classes in ECMAScript have a more limited syntax, and some character class language elements have a different meaning. For example, ECMAScript does not support language elements such as the Unicode category or block elements `\p` and `\P`. Similarly, the `\w` element, which matches a word character, is equivalent to the `[a-zA-Z_0-9]` character class when using ECMAScript and `[\p{L}\p{Lu}\p{Lt}\p{Lo}\p{Nd}\p{Pc}\p{Lm}]` when using canonical behavior. For more information, see [Character Classes](#).

The following example illustrates the difference between canonical and ECMAScript pattern matching. It defines a regular expression, `\b(\w+\s*)+`, that matches words followed by white-space characters. The input consists of two strings, one that uses the Latin character set and the other that uses the Cyrillic character set. As the output shows, the call to the `Regex.IsMatch(String, String, RegexOptions)` method that uses ECMAScript matching fails to match the Cyrillic words, whereas the method call that uses canonical matching does match these words.

```
C#  
  
using System;  
using System.Text.RegularExpressions;  
  
public class EcmaScriptExample  
{  
    public static void Main()  
    {  
        string[] values = { "целый мир", "the whole world" };  
        string pattern = @"\b(\w+\s*)+";  
        foreach (var value in values)  
        {  
            Console.Write("Canonical matching: ");  
            if (Regex.IsMatch(value, pattern))  
                Console.WriteLine("{0} matches the pattern.", value);  
            else  
                Console.WriteLine("{0} does not match the pattern.",  
value);  
  
            Console.Write("ECMAScript matching: ");  
        }  
    }  
}
```

```

        if (Regex.IsMatch(value, pattern, RegexOptions.ECMAScript))
            Console.WriteLine("'{0}' matches the pattern.", value);
        else
            Console.WriteLine("{0} does not match the pattern.",
value);
        Console.WriteLine();
    }
}

// The example displays the following output:
//      Canonical matching: 'целый мир' matches the pattern.
//      ECMAScript matching: целый мир does not match the pattern.
//
//      Canonical matching: 'the whole world' matches the pattern.
//      ECMAScript matching: 'the whole world' matches the pattern.

```

- Self-referencing capturing groups. A regular expression capture class with a backreference to itself must be updated with each capture iteration. As the following example shows, this feature enables the regular expression `((a+)(\1) ?)+` to match the input string " aa aaaa aaaaaa " when using ECMAScript, but not when using canonical matching.

C#

```

using System;
using System.Text.RegularExpressions;

public class EcmaScript2Example
{
    static string pattern;

    public static void Main()
    {
        string input = "aa aaaa aaaaaa ";
        pattern = @"\((a+)(\1) ?)+";

        // Match input using canonical matching.
        AnalyzeMatch(Regex.Match(input, pattern));

        // Match input using ECMAScript.
        AnalyzeMatch(Regex.Match(input, pattern,
RegexOptions.ECMAScript));
    }

    private static void AnalyzeMatch(Match m)
    {
        if (m.Success)
        {
            Console.WriteLine("'{0}' matches {1} at position {2}.",
pattern, m.Value, m.Index);
            int grpCtr = 0;
            foreach (Group grp in m.Groups)

```

```

    {
        Console.WriteLine("    {0}: '{1}'", grpCtr, grp.Value);
        grpCtr++;
        int capCtr = 0;
        foreach (Capture cap in grp.Captures)
        {
            Console.WriteLine("        {0}: '{1}'", capCtr,
cap.Value);
            capCtr++;
        }
    }
    else
    {
        Console.WriteLine("No match found.");
    }
    Console.WriteLine();
}
// The example displays the following output:
// No match found.
//
//     '((a+)(\1) ?)+' matches aa aaaa aaaaaa at position 0.
//         0: 'aa aaaa aaaaaa '
//             0: 'aa aaaa aaaaaa '
//                 1: 'aaaaaa '
//                     0: 'aa '
//                         1: 'aaa '
//                             2: 'aaaaaa '
//                                 2: 'aa'
//                                     0: 'aa'
//                                         1: 'aa'
//                                             2: 'aa'
//                                                 3: 'aaa '
//                                                     0: ''
//                                                         1: 'aa '
//                                                             2: 'aaaa '

```

The regular expression is defined as shown in the following table.

| Pattern | Description |
|----------------|--|
| (a+) | Match the letter "a" one or more times. This is the second capturing group. |
| (\1) | Match the substring captured by the first capturing group. This is the third capturing group. |
| ? | Match zero or one space characters. |
| ((a+)(\1) ?)+ | Match the pattern of one or more "a" characters followed by a string that matches the first capturing group followed by zero or one space characters one or more times. This is the first capturing group. |

- Resolution of ambiguities between octal escapes and backreferences. The following table summarizes the differences in octal versus backreference interpretation by canonical and ECMAScript regular expressions.

| Regular expression | Canonical behavior | ECMAScript behavior |
|--|---|---|
| \0 followed by 0 to 2 octal digits | Interpret as an octal. For example, \044 is always interpreted as an octal value and means "\$". | Same behavior. |
| \ followed by a digit from 1 to 9, followed by no additional decimal digits, | Interpret as a backreference. For example, \9 always means backreference 9, even if a ninth capturing group does not exist. If the capturing group does not exist, the regular expression parser throws an ArgumentException . | If a single decimal digit capturing group exists, backreference to that digit. Otherwise, interpret the value as a literal. |
| \ followed by a digit from 1 to 9, followed by additional decimal digits | Interpret the digits as a decimal value. If that capturing group exists, interpret the expression as a backreference. Otherwise, interpret the leading octal digits up to octal 377; that is, consider only the low 8 bits of the value. Interpret the remaining digits as literals. For example, in the expression \3000, if capturing group 300 exists, interpret as backreference 300; if capturing group 300 does not exist, interpret as octal 300 followed by 0. | Interpret as a backreference by converting as many digits as possible to a decimal value that can refer to a capture. If no digits can be converted, interpret as an octal by using the leading octal digits up to octal 377; interpret the remaining digits as literals. |

Compare using the invariant culture

By default, when the regular expression engine performs case-insensitive comparisons, it uses the casing conventions of the current culture to determine equivalent uppercase and lowercase characters.

However, this behavior is undesirable for some types of comparisons, particularly when comparing user input to the names of system resources, such as passwords, files, or URLs. The following example illustrates such a scenario. The code is intended to block access to any resource whose URL is prefaced with FILE://. The regular expression attempts a case-insensitive match with the string by using the regular expression \$FILE://. However, when the current system culture is tr-TR (Turkish-Türkiye), "I" is not

the uppercase equivalent of "i". As a result, the call to the [Regex.IsMatch](#) method returns `false`, and access to the file is allowed.

C#

```
CultureInfo defaultCulture = Thread.CurrentThread.CurrentCulture;
Thread.CurrentThread.CurrentCulture = new CultureInfo("tr-TR");

string input = "file://c:/Documents.MyReport.doc";
string pattern = "FILE://";

Console.WriteLine("Culture-sensitive matching ({0} culture)...",
                  Thread.CurrentThread.CurrentCulture.Name);
if (Regex.IsMatch(input, pattern, RegexOptions.IgnoreCase))
    Console.WriteLine("URLs that access files are not allowed.");
else
    Console.WriteLine("Access to {0} is allowed.", input);

Thread.CurrentThread.CurrentCulture = defaultCulture;
// The example displays the following output:
//     Culture-sensitive matching (tr-TR culture)...
//     Access to file://c:/Documents.MyReport.doc is allowed.
```

① Note

For more information about string comparisons that are case-sensitive and that use the invariant culture, see [Best Practices for Using Strings](#).

Instead of using the case-insensitive comparisons of the current culture, you can specify the [RegexOptions.CultureInvariant](#) option to ignore cultural differences in language and to use the conventions of the invariant culture.

① Note

Comparison using the invariant culture is available only by supplying the [RegexOptions.CultureInvariant](#) value to the `options` parameter of a [Regex](#) class constructor or static pattern-matching method. It is not available as an inline option.

The following example is identical to the previous example, except that the static [Regex.IsMatch\(String, String, RegexOptions\)](#) method is called with options that include [RegexOptions.CultureInvariant](#). Even when the current culture is set to Turkish (Türkiye), the regular expression engine is able to successfully match "FILE" and "file" and block access to the file resource.

C#

```
CultureInfo defaultCulture = Thread.CurrentThread.CurrentCulture;
Thread.CurrentThread.CurrentCulture = new CultureInfo("tr-TR");

string input = "file:///c:/Documents.MyReport.doc";
string pattern = "FILE://";

Console.WriteLine("Culture-insensitive matching...");
if (Regex.IsMatch(input, pattern,
    RegexOptions.IgnoreCase | RegexOptions.CultureInvariant))
    Console.WriteLine("URLs that access files are not allowed.");
else
    Console.WriteLine("Access to {0} is allowed.", input);

Thread.CurrentThread.CurrentCulture = defaultCulture;
// The example displays the following output:
//     Culture-insensitive matching...
//     URLs that access files are not allowed.
```

Nonbacktracking mode

By default, .NET's regex engine uses *backtracking* to try to find pattern matches. A backtracking engine is one that tries to match one pattern, and if that fails, goes backs and tries to match an alternate pattern, and so on. A backtracking engine is very fast for typical cases, but slows down as the number of pattern alternations increases, which can lead to *catastrophic backtracking*. The [RegexOptions.NonBacktracking](#) option, which was introduced in .NET 7, doesn't use backtracking and avoids that worst-case scenario. Its goal is to provide consistently good behavior, regardless of the input being searched.

The [RegexOptions.NonBacktracking](#) option doesn't support everything the other built-in engines support. In particular, the option can't be used in conjunction with [RegexOptions.RightToLeft](#) or [RegexOptions.ECMAScript](#). It also doesn't allow for the following constructs in the pattern:

- Atomic groups
- Backreferences
- Balancing groups
- Conditionals
- Lookarounds
- Start anchors (`\G`)

[RegexOptions.NonBacktracking](#) also has a subtle difference with regards to execution. If a capture group is in a loop, most (non-.NET) regex engines only provide the last matched value for that capture. However, .NET's regex engine tracks all values that are

captured inside a loop and provides access to them. The `RegexOptions.NonBacktracking` option is like most other regex implementations and only supports providing the final capture.

For more information about backtracking, see [Backtracking in regular expressions](#).

See also

- [Regular Expression Language - Quick Reference](#)

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

Miscellaneous Constructs in Regular Expressions

Article • 10/06/2022

Regular expressions in .NET include three miscellaneous language constructs. One lets you enable or disable particular matching options in the middle of a regular expression pattern. The remaining two let you include comments in a regular expression.

Inline Options

You can set or disable specific pattern matching options for part of a regular expression by using the syntax

```
(?imnsx-imnsx)
```

You list the options you want to enable after the question mark, and the options you want to disable after the minus sign. The following table describes each option. For more information about each option, see [Regular Expression Options](#).

| Option | Description |
|--------|---|
| i | Case-insensitive matching. |
| m | Multiline mode. |
| n | Explicit captures only. (Parentheses do not act as capturing groups.) |
| s | Single-line mode. |
| x | Ignore unescaped white space, and allow x-mode comments. |

Any change in regular expression options defined by the `(?imnsx-imnsx)` construct remains in effect until the end of the enclosing group.

Note

The `(?imnsx-imnsx: subexpression)` grouping construct provides identical functionality for a subexpression. For more information, see [Grouping Constructs](#).

The following example uses the `i`, `n`, and `x` options to enable case insensitivity and explicit captures, and to ignore white space in the regular expression pattern in the

middle of a regular expression.

C#

```
using System;
using System.Text.RegularExpressions;

public class Example
{
    public static void Main()
    {
        string pattern;
        string input = "double dare double Double a Drooling dog The Dreaded
Deep";

        pattern = @"\b(D\w+)\s(d\w+)\b";
        // Match pattern using default options.
        foreach (Match match in Regex.Matches(input, pattern))
        {
            Console.WriteLine(match.Value);
            if (match.Groups.Count > 1)
                for (int ctr = 1; ctr < match.Groups.Count; ctr++)
                    Console.WriteLine("    Group {0}: {1}", ctr,
match.Groups[ctr].Value);
        }
        Console.WriteLine();

        // Change regular expression pattern to include options.
        pattern = @"\b(D\w+)(?ixn) \s (d\w+) \b";
        // Match new pattern with options.
        foreach (Match match in Regex.Matches(input, pattern))
        {
            Console.WriteLine(match.Value);
            if (match.Groups.Count > 1)
                for (int ctr = 1; ctr < match.Groups.Count; ctr++)
                    Console.WriteLine("    Group {0}: '{1}'", ctr,
match.Groups[ctr].Value);
        }
    }
}

// The example displays the following output:
//      Drooling dog
//          Group 1: Drooling
//          Group 2: dog
//
//      Drooling dog
//          Group 1: 'Drooling'
//      Dreaded Deep
//          Group 1: 'Dreaded'
```

The example defines two regular expressions. The first, `\b(D\w+)\s(d\w+)\b`, matches two consecutive words that begin with an uppercase "D" and a lowercase "d". The

second regular expression, `\b(D\w+)(?ixn) \s (d\w+) \b`, uses inline options to modify this pattern, as described in the following table. A comparison of the results confirms the effect of the `(?ixn)` construct.

| Pattern | Description |
|---------------------|---|
| <code>\b</code> | Start at a word boundary. |
| <code>(D\w+)</code> | Match a capital "D" followed by one or more word characters. This is the first capture group. |
| <code>(?ixn)</code> | From this point on, make comparisons case-insensitive, make only explicit captures, and ignore white space in the regular expression pattern. |
| <code>\s</code> | Match a white-space character. |
| <code>(d\w+)</code> | Match an uppercase or lowercase "d" followed by one or more word characters. This group is not captured because the <code>n</code> (explicit capture) option was enabled. |
| <code>\b</code> | Match a word boundary. |

Inline Comment

The `(?# comment)` construct lets you include an inline comment in a regular expression. The regular expression engine does not use any part of the comment in pattern matching, although the comment is included in the string that is returned by the [Regex.ToString](#) method. The comment ends at the first closing parenthesis.

The following example repeats the first regular expression pattern from the example in the previous section. It adds two inline comments to the regular expression to indicate whether the comparison is case-sensitive. The regular expression pattern, `\b((?# case-sensitive comparison)D\w+)\s(?ixn)((?#case-insensitive comparison)d\w+)\b`, is defined as follows.

| Pattern | Description |
|---|---|
| <code>\b</code> | Start at a word boundary. |
| <code>(?# case-sensitive comparison)</code> | A comment. It does not affect pattern-matching behavior. |
| <code>(D\w+)</code> | Match a capital "D" followed by one or more word characters. This is the first capturing group. |
| <code>\s</code> | Match a white-space character. |

| Pattern | Description |
|---------------------------------|---|
| (?ixn) | From this point on, make comparisons case-insensitive, make only explicit captures, and ignore white space in the regular expression pattern. |
| (?#case-insensitive comparison) | A comment. It does not affect pattern-matching behavior. |
| (d\w+) | Match an uppercase or lowercase "d" followed by one or more word characters. This is the second capture group. |
| \b | Match a word boundary. |

C#

```

using System;
using System.Text.RegularExpressions;

public class Example
{
    public static void Main()
    {
        string pattern = @"\b((?# case-sensitive comparison)D\w+)\s(?ixn)((?#
#case-insensitive comparison)d\w+)\b";
        Regex rgx = new Regex(pattern);
        string input = "double dare double Double a Drooling dog The Dreaded
Deep";

        Console.WriteLine("Pattern: " + pattern.ToString());
        // Match pattern using default options.
        foreach (Match match in rgx.Matches(input))
        {
            Console.WriteLine(match.Value);
            if (match.Groups.Count > 1)
            {
                for (int ctr = 1; ctr < match.Groups.Count; ctr++)
                    Console.WriteLine("    Group {0}: {1}", ctr,
match.Groups[ctr].Value);
            }
        }
    }
}
// The example displays the following output:
//      Pattern: \b((?# case-sensitive comparison)D\w+)\s(?ixn)((?#
#case-
insensitive comp
//      arison)d\w+)\b
//      Drooling dog
//          Group 1: Drooling
//      Dreaded Deep
//          Group 1: Dreaded

```

End-of-Line Comment

A number sign (#) marks an x-mode comment, which starts at the unescaped # character at the end of the regular expression pattern and continues until the end of the line. To use this construct, you must either enable the `x` option (through inline options) or supply the `RegexOptions.IgnorePatternWhitespace` value to the `option` parameter when instantiating the `Regex` object or calling a static `Regex` method.

The following example illustrates the end-of-line comment construct. It determines whether a string is a composite format string that includes at least one format item. The following table describes the constructs in the regular expression pattern:

```
\{\ \d+(-*\d+)*(\:\w{1,4}?)*\}\(?x) # Looks for a composite format item.
```

| Pattern | Description |
|---|--|
| <code>\{</code> | Match an opening brace. |
| <code>\d+</code> | Match one or more decimal digits. |
| <code>(,-*\d+)*</code> | Match zero or one occurrence of a comma, followed by an optional minus sign, followed by one or more decimal digits. |
| <code>(\:\w{1,4}?)*</code> | Match zero or one occurrence of a colon, followed by one to four, but as few as possible, white-space characters. |
| <code>\}</code> | Match a closing brace. |
| <code>(?x)</code> | Enable the ignore pattern white-space option so that the end-of-line comment will be recognized. |
| <code># Looks for a composite format item.</code> | An end-of-line comment. |

C#

```
using System;
using System.Text.RegularExpressions;

public class Example
{
    public static void Main()
    {
        string pattern = @"\{\ \d+(-*\d+)*(\:\w{1,4}?)*\}\(?x) # Looks for a
composite format item.";
        string input = "{0,-3:F}";
        Console.WriteLine('{0}:', input);
        if (Regex.IsMatch(input, pattern))
            Console.WriteLine("    contains a composite format item.");
    }
}
```

```
        else
            Console.WriteLine("    does not contain a composite format item.");
    }
}
// The example displays the following output:
//      '{0,-3:F}':
//      contains a composite format item.
```

Note that, instead of providing the `(?x)` construct in the regular expression, the comment could also have been recognized by calling the [Regex.IsMatch\(String, String, RegexOptions\)](#) method and passing it the [RegexOptions.IgnorePatternWhitespace](#) enumeration value.

See also

- [Regular Expression Language - Quick Reference](#)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

Best practices for regular expressions in .NET

Article • 10/03/2023

The regular expression engine in .NET is a powerful, full-featured tool that processes text based on pattern matches rather than on comparing and matching literal text. In most cases, it performs pattern matching rapidly and efficiently. However, in some cases, the regular expression engine can appear to be slow. In extreme cases, it can even appear to stop responding as it processes a relatively small input over the course of hours or even days.

This article outlines some of the best practices that developers can adopt to ensure that their regular expressions achieve optimal performance.

Warning

When using `System.Text.RegularExpressions` to process untrusted input, pass a timeout. A malicious user can provide input to `RegularExpressions`, causing a [Denial-of-Service attack ↗](#). ASP.NET Core framework APIs that use `RegularExpressions` pass a timeout.

Consider the input source

In general, regular expressions can accept two types of input: constrained or unconstrained. Constrained input is a text that originates from a known or reliable source and follows a predefined format. Unconstrained input is a text that originates from an unreliable source, such as a web user, and might not follow a predefined or expected format.

Regular expression patterns are often written to match valid input. That is, developers examine the text that they want to match and then write a regular expression pattern that matches it. Developers then determine whether this pattern requires correction or further elaboration by testing it with multiple valid input items. When the pattern matches all presumed valid inputs, it's declared to be production-ready, and can be included in a released application. This approach makes a regular expression pattern suitable for matching constrained input. However, it doesn't make it suitable for matching unconstrained input.

To match unconstrained input, a regular expression must handle three kinds of text efficiently:

- Text that matches the regular expression pattern.
- Text that doesn't match the regular expression pattern.
- Text that nearly matches the regular expression pattern.

The last text type is especially problematic for a regular expression that has been written to handle constrained input. If that regular expression also relies on extensive [backtracking](#), the regular expression engine can spend an inordinate amount of time (in some cases, many hours or days) processing seemingly innocuous text.

Warning

The following example uses a regular expression that's prone to excessive backtracking and that's likely to reject valid email addresses. You shouldn't use it in an email validation routine. If you would like a regular expression that validates email addresses, see [How to: Verify that Strings Are in Valid Email Format](#).

For example, consider a commonly used but problematic regular expression for validating the alias of an email address. The regular expression `@[0-9A-Z]([-.\\w]*[0-9A-Z])*$` is written to process what is considered to be a valid email address. A valid email address consists of an alphanumeric character, followed by zero or more characters that can be alphanumeric, periods, or hyphens. The regular expression must end with an alphanumeric character. However, as the following example shows, although this regular expression handles valid input easily, its performance is inefficient when it's processing nearly valid input:

C#

```
using System;
using System.Diagnostics;
using System.Text.RegularExpressions;

public class Example
{
    public static void Main()
    {
        Stopwatch sw;
        string[] addresses = { "AAAAAAA@contoso.com",
                               "AAAAAAAAAaaaaaaaaa!@contoso.com" };
        // The following regular expression should not actually be used to
        // validate an email address.
        string pattern = @"@[0-9A-Z]([-.\\w]*[0-9A-Z])*$";
        string input;
```

```

        foreach (var address in addresses) {
            string mailBox = address.Substring(0, address.IndexOf("@"));
            int index = 0;
            for (int ctr = mailBox.Length - 1; ctr >= 0; ctr--) {
                index++;

                input = mailBox.Substring(ctr, index);
                sw = Stopwatch.StartNew();
                Match m = Regex.Match(input, pattern, RegexOptions.IgnoreCase);
                sw.Stop();
                if (m.Success)
                    Console.WriteLine("{0,2}. Matched '{1,25}' in {2}",
                                      index, m.Value, sw.Elapsed);
                else
                    Console.WriteLine("{0,2}. Failed  '{1,25}' in {2}",
                                      index, input, sw.Elapsed);
            }
            Console.WriteLine();
        }
    }

// The example displays output similar to the following:
//      1. Matched '          A' in 00:00:00.0007122
//      2. Matched '         AA' in 00:00:00.0000282
//      3. Matched '        AAA' in 00:00:00.0000042
//      4. Matched '       AAAA' in 00:00:00.0000038
//      5. Matched '      AAAAA' in 00:00:00.0000042
//      6. Matched '     AAAAAA' in 00:00:00.0000042
//      7. Matched '    AAAAAAA' in 00:00:00.0000042
//      8. Matched '   AAAAAAAA' in 00:00:00.0000087
//      9. Matched '  AAAAAAAAA' in 00:00:00.0000045
//     10. Matched ' AAAAAAAAAA' in 00:00:00.0000045
//     11. Matched 'AAAAAAA" in 00:00:00.0000045
//
//      1. Failed  '          !' in 00:00:00.0000447
//      2. Failed  '         a!' in 00:00:00.0000071
//      3. Failed  '        aa!' in 00:00:00.0000071
//      4. Failed  '       aaa!' in 00:00:00.0000061
//      5. Failed  '      aaaa!' in 00:00:00.0000081
//      6. Failed  '     aaaaa!' in 00:00:00.0000126
//      7. Failed  '    aaaaaa!' in 00:00:00.0000359
//      8. Failed  '   aaaaaaa!' in 00:00:00.0000414
//      9. Failed  '  aaaaaaaa!' in 00:00:00.0000758
//     10. Failed  ' aaaaaaaaa!' in 00:00:00.0001462
//     11. Failed  'aaaaaaa" in 00:00:00.0002885
//     12. Failed  'Aaaaaaaaaaa!' in 00:00:00.0005780
//     13. Failed  'AAaaaaaaaaaa!' in 00:00:00.0011628
//     14. Failed  'AAAaaaaaaaaaa!' in 00:00:00.0022851
//     15. Failed  'AAAAaaaaaaaaaa!' in 00:00:00.0045864
//     16. Failed  'AAAAAaaaaaaaaaa!' in 00:00:00.0093168
//     17. Failed  'AAAAAAaaaaaaaaaa!' in 00:00:00.0185993
//     18. Failed  'AAAAAAAaaaaaaaaaa!' in 00:00:00.0366723
//     19. Failed  'AAAAAAAaaaaaaaaaa!' in 00:00:00.1370108

```

```
// 20. Failed 'AAAAAAAAaaaaaaa!' in 00:00:00.1553966
// 21. Failed 'AAAAAAAAAaaaaaaaa!' in 00:00:00.3223372
```

As the output from the preceding example shows, the regular expression engine processes the valid email alias in about the same time interval regardless of its length. On the other hand, when the nearly valid email address has more than five characters, processing time approximately doubles for each extra character in the string. Therefore, a nearly valid 28-character string would take over an hour to process, and a nearly valid 33-character string would take nearly a day to process.

Because this regular expression was developed solely by considering the format of input to be matched, it fails to take account of input that doesn't match the pattern. This oversight, in turn, can allow unconstrained input that nearly matches the regular expression pattern to significantly degrade performance.

To solve this problem, you can do the following:

- When developing a pattern, you should consider how backtracking might affect the performance of the regular expression engine, particularly if your regular expression is designed to process unconstrained input. For more information, see the [Take Charge of Backtracking](#) section.
- Thoroughly test your regular expression using invalid, near-valid, and valid input. You can use [Rex ↗](#) to randomly generate input for a particular regular expression. [Rex ↗](#) is a regular expression exploration tool from Microsoft Research.

Handle object instantiation appropriately

At the heart of .NET's regular expression object model, is the [System.Text.RegularExpressions.Regex](#) class, which represents the regular expression engine. Often, the single greatest factor that affects regular expression performance is the way in which the [Regex](#) engine is used. Defining a regular expression involves tightly coupling the regular expression engine with a regular expression pattern. That coupling process, whether it involves instantiating a [Regex](#) object by passing its constructor a regular expression pattern or calling a static method by passing it the regular expression pattern and the string to be analyzed, is by necessity an expensive one.

ⓘ Note

For a detailed discussion of the performance implications of using interpreted and compiled regular expressions, see [Optimizing Regular Expression Performance, Part II: Taking Charge of Backtracking](#) in the BCL Team blog.

You can couple the regular expression engine with a particular regular expression pattern and then use the engine to match the text in several ways:

- You can call a static pattern-matching method, such as [Regex.Match\(String, String\)](#). This method doesn't require instantiation of a regular expression object.
- You can instantiate a [Regex](#) object and call an instance pattern-matching method of an interpreted regular expression, which is the default method for binding the regular expression engine to a regular expression pattern. It results when a [Regex](#) object is instantiated without an `options` argument that includes the [Compiled](#) flag.
- You can instantiate a [Regex](#) object and call an instance pattern-matching method of a compiled regular expression. Regular expression objects represent compiled patterns when a [Regex](#) object is instantiated with an `options` argument that includes the [Compiled](#) flag.
- You can create a special-purpose [Regex](#) object that's tightly coupled with a particular regular expression pattern, compile it, and save it to a standalone assembly. You can call the [Regex.CompileToAssembly](#) method to compile and save it.

The particular way in which you call regular expression matching methods can affect your application's performance. The following sections discuss when to use static method calls, interpreted regular expressions, and compiled regular expressions to improve your application's performance.

 **Important**

The form of the method call (static, interpreted, compiled) affects performance if the same regular expression is used repeatedly in method calls, or if an application makes extensive use of regular expression objects.

Static regular expressions

Static regular expression methods are recommended as an alternative to repeatedly instantiating a regular expression object with the same regular expression. Unlike

regular expression patterns used by regular expression objects, either the operation codes or the compiled Microsoft intermediate language (MSIL) from patterns used in static method calls is cached internally by the regular expression engine.

For example, an event handler frequently calls another method to validate user input. This example is reflected in the following code, in which a [Button](#) control's [Click](#) event is used to call a method named `IsValidCurrency`, which checks whether the user has entered a currency symbol followed by at least one decimal digit.

C#

```
public void OKButton_Click(object sender, EventArgs e)
{
    if (! String.IsNullOrEmpty(sourceCurrency.Text))
        if (RegexLib.IsValidCurrency(sourceCurrency.Text))
            PerformConversion();
        else
            status.Text = "The source currency value is invalid.";
}
```

An inefficient implementation of the `IsValidCurrency` method is shown in the following example:

① Note

Each method call reinstatiates a [Regex](#) object with the same pattern. This, in turn, means that the regular expression pattern must be recompiled each time the method is called.

C#

```
using System;
using System.Text.RegularExpressions;

public class RegexLib
{
    public static bool IsValidCurrency(string currencyValue)
    {
        string pattern = @"\p{Sc}+\s*\d+";
        Regex currencyRegex = new Regex(pattern);
        return currencyRegex.IsMatch(currencyValue);
    }
}
```

You should replace the preceding inefficient code with a call to the static [Regex.IsMatch\(String, String\)](#) method. This approach eliminates the need to instantiate a

[Regex](#) object each time you want to call a pattern-matching method, and enables the regular expression engine to retrieve a compiled version of the regular expression from its cache.

```
C#  
  
using System;  
using System.Text.RegularExpressions;  
  
public class RegexLib  
{  
    public static bool IsValidCurrency(string currencyValue)  
    {  
        string pattern = @"\p{Sc}+\s*\d+";  
        return Regex.IsMatch(currencyValue, pattern);  
    }  
}
```

By default, the last 15 most recently used static regular expression patterns are cached. For applications that require a larger number of cached static regular expressions, the size of the cache can be adjusted by setting the [Regex.CacheSize](#) property.

The regular expression `\p{Sc}+\s*\d+` that's used in this example verifies that the input string has a currency symbol and at least one decimal digit. The pattern is defined as shown in the following table:

| Pattern | Description |
|----------------------|--|
| <code>\p{Sc}+</code> | Matches one or more characters in the Unicode Symbol, Currency category. |
| <code>\s*</code> | Matches zero or more white-space characters. |
| <code>\d+</code> | Matches one or more decimal digits. |

Interpreted vs. compiled regular expressions

Regular expression patterns that aren't bound to the regular expression engine through the specification of the [Compiled](#) option are interpreted. When a regular expression object is instantiated, the regular expression engine converts the regular expression to a set of operation codes. When an instance method is called, the operation codes are converted to MSIL and executed by the JIT compiler. Similarly, when a static regular expression method is called and the regular expression can't be found in the cache, the regular expression engine converts the regular expression to a set of operation codes and stores them in the cache. It then converts these operation codes to MSIL so that the JIT compiler can execute them. Interpreted regular expressions reduce startup time at

the cost of slower execution time. Because of this process, they're best used when the regular expression is used in a small number of method calls, or if the exact number of calls to regular expression methods is unknown but is expected to be small. As the number of method calls increases, the performance gain from reduced startup time is outstripped by the slower execution speed.

Regular expression patterns that are bound to the regular expression engine through the specification of the [Compiled](#) option are compiled. Therefore, when a regular expression object is instantiated, or when a static regular expression method is called and the regular expression can't be found in the cache, the regular expression engine converts the regular expression to an intermediary set of operation codes. These codes are then converted to MSIL. When a method is called, the JIT compiler executes the MSIL. In contrast to interpreted regular expressions, compiled regular expressions increase startup time but execute individual pattern-matching methods faster. As a result, the performance benefit that results from compiling the regular expression increases in proportion to the number of regular expression methods called.

To summarize, we recommend that you use interpreted regular expressions when you call regular expression methods with a specific regular expression relatively infrequently. You should use compiled regular expressions when you call regular expression methods with a specific regular expression relatively frequently. It's difficult to determine the exact threshold at which the slower execution speeds of interpreted regular expressions outweigh gains from their reduced startup time, or the threshold at which the slower startup times of compiled regular expressions outweigh gains from their faster execution speeds. It depends on various factors, including the complexity of the regular expression and the specific data that it processes. To determine whether interpreted or compiled regular expressions offer the best performance for your particular application scenario, you can use the [Stopwatch](#) class to compare their execution times.

The following example compares the performance of compiled and interpreted regular expressions when reading the first 10 sentences and when reading all the sentences in the text of Theodore Dreiser's *The Financier*. As the output from the example shows, when only 10 calls are made to regular expression matching methods, an interpreted regular expression offers better performance than a compiled regular expression. However, a compiled regular expression offers better performance when a large number of calls (in this case, over 13,000) are made.

C#

```
using System;
using System.Diagnostics;
using System.IO;
using System.Text.RegularExpressions;
```

```
public class Example
{
    public static void Main()
    {
        string pattern = @"\b(\w+((\r?\n)|,\s))*\w+[.?;!]";
        Stopwatch sw;
        Match match;
        int ctr;

        StreamReader inFile = new StreamReader(@".\Dreiser_TheFinancier.txt");
        string input = inFile.ReadToEnd();
        inFile.Close();

        // Read first ten sentences with interpreted regex.
        Console.WriteLine("10 Sentences with Interpreted Regex:");
        sw = Stopwatch.StartNew();
        Regex int10 = new Regex(pattern, RegexOptions.Singleline);
        match = int10.Match(input);
        for (ctr = 0; ctr <= 9; ctr++) {
            if (match.Success)
                // Do nothing with the match except get the next match.
                match = match.NextMatch();
            else
                break;
        }
        sw.Stop();
        Console.WriteLine("    {0} matches in {1}", ctr, sw.Elapsed);

        // Read first ten sentences with compiled regex.
        Console.WriteLine("10 Sentences with Compiled Regex:");
        sw = Stopwatch.StartNew();
        Regex comp10 = new Regex(pattern,
                                RegexOptions.Singleline | RegexOptions.Compiled);
        match = comp10.Match(input);
        for (ctr = 0; ctr <= 9; ctr++) {
            if (match.Success)
                // Do nothing with the match except get the next match.
                match = match.NextMatch();
            else
                break;
        }
        sw.Stop();
        Console.WriteLine("    {0} matches in {1}", ctr, sw.Elapsed);

        // Read all sentences with interpreted regex.
        Console.WriteLine("All Sentences with Interpreted Regex:");
        sw = Stopwatch.StartNew();
        Regex intAll = new Regex(pattern, RegexOptions.Singleline);
        match = intAll.Match(input);
        int matches = 0;
        while (match.Success) {
            matches++;
            // Do nothing with the match except get the next match.
            match = match.NextMatch();
        }
    }
}
```

```

        }
        sw.Stop();
        Console.WriteLine("    {0:N0} matches in {1}", matches, sw.Elapsed);

        // Read all sentences with compiled regex.
        Console.WriteLine("All Sentences with Compiled Regex:");
        sw = Stopwatch.StartNew();
        Regex compAll = new Regex(pattern,
            RegexOptions.Singleline | RegexOptions.Compiled);
        match = compAll.Match(input);
        matches = 0;
        while (match.Success) {
            matches++;
            // Do nothing with the match except get the next match.
            match = match.NextMatch();
        }
        sw.Stop();
        Console.WriteLine("    {0:N0} matches in {1}", matches, sw.Elapsed);
    }
}

// The example displays the following output:
//      10 Sentences with Interpreted Regex:
//          10 matches in 00:00:00.0047491
//      10 Sentences with Compiled Regex:
//          10 matches in 00:00:00.0141872
//      All Sentences with Interpreted Regex:
//          13,443 matches in 00:00:01.1929928
//      All Sentences with Compiled Regex:
//          13,443 matches in 00:00:00.7635869

//
//      >compare1
//      10 Sentences with Interpreted Regex:
//          10 matches in 00:00:00.0046914
//      10 Sentences with Compiled Regex:
//          10 matches in 00:00:00.0143727
//      All Sentences with Interpreted Regex:
//          13,443 matches in 00:00:01.1514100
//      All Sentences with Compiled Regex:
//          13,443 matches in 00:00:00.7432921

```

The regular expression pattern used in the example, `\b(\w+((\r?\n)|,\s))*\w+[.?:;!]`, is defined as shown in the following table:

| Pattern | Description |
|-------------------------|---|
| <code>\b</code> | Begin the match at a word boundary. |
| <code>\w+</code> | Matches one or more word characters. |
| <code>(\r?\n) ,?</code> | Matches either zero or one carriage return followed by a newline character, or zero or one comma followed by a white-space character. |
| <code>\s</code> | |

| Pattern | Description |
|----------------------|---|
| (\w+((\r?\n) ,?\s))* | Matches zero or more occurrences of one or more word characters that are followed either by zero or one carriage return and a newline character, or by zero or one comma followed by a white-space character. |
| \w+ | Matches one or more word characters. |
| [.?;:] | Matches a period, question mark, colon, semicolon, or exclamation point. |

Regular expressions: Compiled to an assembly

.NET also enables you to create an assembly that contains compiled regular expressions. This capability moves the performance hit of regular expression compilation from run time to design time. However, it also involves some additional work. You must define the regular expressions in advance and compile them to an assembly. The compiler can then reference this assembly when compiling source code that uses the assembly's regular expressions. Each compiled regular expression in the assembly is represented by a class that derives from [Regex](#).

To compile regular expressions to an assembly, you call the [Regex.CompileToAssembly\(RegexCompilationInfo\[\], AssemblyName\)](#) method and pass it an array of [RegexCompilationInfo](#) objects and an [AssemblyName](#) object. The [RegexCompilationInfo](#) objects represent the regular expressions to be compiled, and the [AssemblyName](#) object that contains information about the assembly to be created.

We recommend that you compile regular expressions to an assembly in the following situations:

- If you're a component developer who wants to create a library of reusable regular expressions.
- If you expect your regular expression's pattern-matching methods to be called an indeterminate number of times—anywhere from once or twice to thousands or tens of thousands of times. Unlike compiled or interpreted regular expressions, regular expressions that are compiled to separate assemblies offer performance that's consistent regardless of the number of method calls.

If you're using compiled regular expressions to optimize performance, you shouldn't use reflection to create the assembly, load the regular expression engine, and execute its pattern-matching methods. Avoiding reflection requires that you don't build regular expression patterns dynamically, and that you specify any pattern-matching options, such as case-insensitive pattern matching, at the time the assembly is created. It also

requires that you separate the code that creates the assembly from the code that uses the regular expression.

The following example shows how to create an assembly that contains a compiled regular expression. It creates an assembly named `RegexLib.dll` with a single regular expression class, `SentencePattern`. This class contains the sentence-matching regular expression pattern used in the [Interpreted vs. Compiled Regular Expressions](#) section.

```
C#  
  
using System;  
using System.Reflection;  
using System.Text.RegularExpressions;  
  
public class Example  
{  
    public static void Main()  
    {  
        RegexCompilationInfo SentencePattern =  
            new RegexCompilationInfo(@"\b(\w+((\r?\n)|,?  
\s))*\w+[.:;!]",  
                RegexOptions.Multiline,  
                "SentencePattern",  
                "Utilities.RegularExpressions",  
                true);  
        RegexCompilationInfo[] regexes = { SentencePattern };  
        AssemblyName assemName = new AssemblyName("RegexLib,  
Version=1.0.0.1001, Culture=neutral, PublicKeyToken=null");  
        Regex.CompileToAssembly(regexes, assemName);  
    }  
}
```

When the example is compiled to an executable and run, it creates an assembly named `RegexLib.dll`. A `Utilities.RegularExpressions.SentencePattern` class derived from `Regex` represents the regular expression. The following example then uses the compiled regular expression to extract the sentences from the text of Theodore Dreiser's *The Financier*:

```
C#  
  
using System;  
using System.IO;  
using System.Text.RegularExpressions;  
using Utilities.RegularExpressions;  
  
public class Example  
{  
    public static void Main()  
    {
```

```
{  
    SentencePattern pattern = new SentencePattern();  
    StreamReader inFile = new StreamReader(@".\Dreiser_TheFinancier.txt");  
    string input = inFile.ReadToEnd();  
    inFile.Close();  
  
    MatchCollection matches = pattern.Matches(input);  
    Console.WriteLine("Found {0:N0} sentences.", matches.Count);  
}  
}  
// The example displays the following output:  
//      Found 13,443 sentences.
```

Take charge of backtracking

Ordinarily, the regular expression engine uses linear progression to move through an input string and compare it to a regular expression pattern. However, when indeterminate quantifiers such as `*`, `+`, and `?` are used in a regular expression pattern, the regular expression engine might give up a portion of successful partial matches and return to a previously saved state in order to search for a successful match for the entire pattern. This process is known as backtracking.

Tip

For more information on backtracking, see [Details of regular expression behavior](#) and [Backtracking](#). For detailed discussions of backtracking, see the [Regular Expression Improvements in .NET 7](#) and [Optimizing Regular Expression Performance](#) blog posts.

Support for backtracking gives regular expressions power and flexibility. It also places the responsibility for controlling the operation of the regular expression engine in the hands of regular expression developers. Because developers are often not aware of this responsibility, their misuse of backtracking or reliance on excessive backtracking often plays the most significant role in degrading regular expression performance. In a worst-case scenario, execution time can double for each additional character in the input string. In fact, by using backtracking excessively, it's easy to create the programmatic equivalent of an endless loop if input nearly matches the regular expression pattern. The regular expression engine might take hours or even days to process a relatively short input string.

Often, applications pay a performance penalty for using backtracking even though backtracking isn't essential for a match. For example, the regular expression

`\b\p{Lu}\w*\b` matches all words that begin with an uppercase character, as the following table shows:

| Pattern | Description |
|---------------------|---------------------------------------|
| <code>\b</code> | Begin the match at a word boundary. |
| <code>\p{Lu}</code> | Matches an uppercase character. |
| <code>\w*</code> | Matches zero or more word characters. |
| <code>\b</code> | End the match at a word boundary. |

Because a word boundary isn't the same as, or a subset of, a word character, there's no possibility that the regular expression engine will cross a word boundary when matching word characters. Therefore for this regular expression, backtracking can never contribute to the overall success of any match. It can only degrade performance because the regular expression engine is forced to save its state for each successful preliminary match of a word character.

If you determine that backtracking isn't necessary, you can disable it in a couple of ways:

- By setting the [RegexOptions.NonBacktracking](#) option (introduced in .NET 7). For more information, see [Nonbacktracking mode](#).
- By using the `(?>subexpression)` language element, known as an atomic group. The following example parses an input string by using two regular expressions. The first, `\b\p{Lu}\w*\b`, relies on backtracking. The second, `\b\p{Lu}(?>\w*)\b`, disables backtracking. As the output from the example shows, they both produce the same result:

```
C#  
  
using System;  
using System.Text.RegularExpressions;  
  
public class Example  
{  
    public static void Main()  
    {  
        string input = "This this word Sentence name Capital";  
        string pattern = @"\b\p{Lu}\w*\b";  
        foreach (Match match in Regex.Matches(input, pattern))  
            Console.WriteLine(match.Value);  
  
        Console.WriteLine();  
  
        pattern = @"\b\p{Lu}(?>\w*)\b";
```

```

        foreach (Match match in Regex.Matches(input, pattern))
            Console.WriteLine(match.Value);
    }
}

// The example displays the following output:
//      This
//      Sentence
//      Capital
//
//      This
//      Sentence
//      Capital

```

In many cases, backtracking is essential for matching a regular expression pattern to input text. However, excessive backtracking can severely degrade performance and create the impression that an application has stopped responding. In particular, this problem arises when quantifiers are nested and the text that matches the outer subexpression is a subset of the text that matches the inner subexpression.

Warning

In addition to avoiding excessive backtracking, you should use the timeout feature to ensure that excessive backtracking doesn't severely degrade regular expression performance. For more information, see the [Use time-out values](#) section.

For example, the regular expression pattern `^[0-9A-Z]([-.\\w]*[0-9A-Z])*\\$` is intended to match a part number that consists of at least one alphanumeric character. Any additional characters can consist of an alphanumeric character, a hyphen, an underscore, or a period, though the last character must be alphanumeric. A dollar sign terminates the part number. In some cases, this regular expression pattern can exhibit poor performance because quantifiers are nested, and because the subexpression `[0-9A-Z]` is a subset of the subexpression `[-.\\w]*`.

In these cases, you can optimize regular expression performance by removing the nested quantifiers and replacing the outer subexpression with a zero-width lookahead or lookbehind assertion. Lookahead and lookbehind assertions are anchors. They don't move the pointer in the input string but instead look ahead or behind to check whether a specified condition is met. For example, the part number regular expression can be rewritten as `^[0-9A-Z][-.\\w]*(?<=[0-9A-Z])\\$`. This regular expression pattern is defined as shown in the following table:

| Pattern | Description |
|----------------|---|
| <code>^</code> | Begin the match at the beginning of the input string. |

| Pattern | Description |
|----------------|---|
| [0-9A-Z] | Match an alphanumeric character. The part number must consist of at least this character. |
| [-.\w]* | Match zero or more occurrences of any word character, hyphen, or period. |
| \\$ | Match a dollar sign. |
| (?<=[0-9A-Z]) | Look behind the ending dollar sign to ensure that the previous character is alphanumeric. |
| \$ | End the match at the end of the input string. |

The following example illustrates the use of this regular expression to match an array containing possible part numbers:

```
C#
using System;
using System.Text.RegularExpressions;

public class Example
{
    public static void Main()
    {
        string pattern = @"^([0-9A-Z][ -.\w]*(?<=[0-9A-Z]))\$$";
        string[] partNos = { "A1C$", "A4", "A4$", "A1603D$", "A1603D#" };

        foreach (var input in partNos) {
            Match match = Regex.Match(input, pattern);
            if (match.Success)
                Console.WriteLine(match.Value);
            else
                Console.WriteLine("Match not found.");
        }
    }
}
// The example displays the following output:
//      A1C$
//      Match not found.
//      A4$
//      A1603D$
//      Match not found.
```

The regular expression language in .NET includes the following language elements that you can use to eliminate nested quantifiers. For more information, see [Grouping constructs](#).

| Language element | Description |
|--------------------------------------|--|
| <code>(?= subexpression)</code> | Zero-width positive lookahead. Looks ahead of the current position to determine whether <code>subexpression</code> matches the input string. |
| <code>(?! subexpression)</code> | Zero-width negative lookahead. Looks ahead of the current position to determine whether <code>subexpression</code> doesn't match the input string. |
| <code>(?=< subexpression)</code> | Zero-width positive lookbehind. Looks behind the current position to determine whether <code>subexpression</code> matches the input string. |
| <code>(?<! subexpression)</code> | Zero-width negative lookbehind. Looks behind the current position to determine whether <code>subexpression</code> doesn't match the input string. |

Use time-out values

If your regular expressions processes input that nearly matches the regular expression pattern, it can often rely on excessive backtracking, which impacts its performance significantly. In addition to carefully considering your use of backtracking and testing the regular expression against near-matching input, you should always set a time-out value to minimize the effect of excessive backtracking, if it occurs.

The regular expression time-out interval defines the period of time that the regular expression engine will look for a single match before it times out. Depending on the regular expression pattern and the input text, the execution time might exceed the specified time-out interval, but it won't spend more time backtracking than the specified time-out interval. The default time-out interval is [Regex.InfiniteMatchTimeout](#), which means that the regular expression won't time out. You can override this value and define a time-out interval as follows:

- Call the [Regex\(String, RegexOptions, TimeSpan\)](#) constructor to provide a time-out value when you instantiate a [Regex](#) object.
- Call a static pattern matching method, such as [Regex.Match\(String, String, RegexOptions, TimeSpan\)](#) or [Regex.Replace\(String, String, String, RegexOptions, TimeSpan\)](#), that includes a `matchTimeout` parameter.
- Call the constructor that has a parameter of type [TimeSpan](#) for compiled regular expressions that are created by calling the [Regex.CompileToAssembly](#) method.
- Set a process-wide or AppDomain-wide value with code such as

```
AppDomain.CurrentDomain.SetData("REGEX_DEFAULT_MATCH_TIMEOUT",
    TimeSpan.FromMilliseconds(100));
```

If you've defined a time-out interval and a match isn't found at the end of that interval, the regular expression method throws a [RegexMatchTimeoutException](#) exception. In your exception handler, you can choose to retry the match with a longer time-out interval, abandon the match attempt and assume that there's no match, or abandon the match attempt and log the exception information for future analysis.

The following example defines a `GetWordData` method that instantiates a regular expression with a time-out interval of 350 milliseconds to calculate the number of words and average number of characters in a word in a text document. If the matching operation times out, the time-out interval is increased by 350 milliseconds and the `Regex` object is reinstated. If the new time-out interval exceeds one second, the method rethrows the exception to the caller.

C#

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Text.RegularExpressions;

public class Example
{
    public static void Main()
    {
        RegexUtilities util = new RegexUtilities();
        string title = "Doyle - The Hound of the Baskervilles.txt";
        try {
            var info = util.GetWordData(title);
            Console.WriteLine("Words: {0:N0}", info.Item1);
            Console.WriteLine("Average Word Length: {0:N2} characters",
info.Item2);
        }
        catch (IOException e) {
            Console.WriteLine("IOException reading file '{0}'", title);
            Console.WriteLine(e.Message);
        }
        catch (RegexMatchTimeoutException e) {
            Console.WriteLine("The operation timed out after {0:N0}
milliseconds",
e.MatchTimeout.TotalMilliseconds);
        }
    }

    public class RegexUtilities
    {
        public Tuple<int, double> GetWordData(string filename)
        {
            const int MAX_TIMEOUT = 1000; // Maximum timeout interval in
milliseconds.
            const int INCREMENT = 350; // Milliseconds increment of timeout.
```

```
    List<string> exclusions = new List<string>( new string[] { "a", "an",
"the" });
    int[] wordLengths = new int[29];           // Allocate an array of more
than ample size.
    string input = null;
    StreamReader sr = null;
    try {
        sr = new StreamReader(filename);
        input = sr.ReadToEnd();
    }
    catch (FileNotFoundException e) {
        string msg = String.Format("Unable to find the file '{0}'",
filename);
        throw new IOException(msg, e);
    }
    catch (IOException e) {
        throw new IOException(e.Message, e);
    }
    finally {
        if (sr != null) sr.Close();
    }

    int timeoutInterval = INCREMENT;
    bool init = false;
    Regex rgx = null;
    Match m = null;
    int indexPos = 0;
    do {
        try {
            if (! init) {
                rgx = new Regex(@"\b\w+\b", RegexOptions.None,
                    TimeSpan.FromMilliseconds(timeoutInterval));
                m = rgx.Match(input, indexPos);
                init = true;
            }
            else {
                m = m.NextMatch();
            }
            if (m.Success) {
                if ( !exclusions.Contains(m.Value.ToLower()))
                    wordLengths[m.Value.Length]++;
            }
            indexPos += m.Length + 1;
        }
    }
    catch (RegexMatchTimeoutException e) {
        if (e.MatchTimeout.TotalMilliseconds < MAX_TIMEOUT) {
            timeoutInterval += INCREMENT;
            init = false;
        }
        else {
            // Rethrow the exception.
            throw;
        }
    }
}
```

```

        }

    } while (m.Success);

    // If regex completed successfully, calculate number of words and
    // average length.
    int nWords = 0;
    long totalLength = 0;

    for (int ctr = wordLengths.GetLowerBound(0); ctr <=
wordLengths.GetUpperBound(0); ctr++) {
        nWords += wordLengths[ctr];
        totalLength += ctr * wordLengths[ctr];
    }
    return new Tuple<int, double>(nWords, totalLength/nWords);
}
}

```

Capture only when necessary

Regular expressions in .NET support grouping constructs, which let you group a regular expression pattern into one or more subexpressions. The most commonly used grouping constructs in .NET regular expression language are `(subexpression)`, which defines a numbered capturing group, and `(?name)subexpression`, which defines a named capturing group. Grouping constructs are essential for creating backreferences and for defining a subexpression to which a quantifier is applied.

However, the use of these language elements has a cost. They cause the [GroupCollection](#) object returned by the [Match.Groups](#) property to be populated with the most recent unnamed or named captures. If a single grouping construct has captured multiple substrings in the input string, they also populate the [CaptureCollection](#) object returned by the [Group.Captures](#) property of a particular capturing group with multiple [Capture](#) objects.

Often, grouping constructs are used in a regular expression only so that quantifiers can be applied to them. The groups captured by these subexpressions aren't used later. For example, the regular expression `\b(\w+[,]?|\s?)+[.?!]` is designed to capture an entire sentence. The following table describes the language elements in this regular expression pattern and their effect on the [Match](#) object's [Match.Groups](#) and [Group.Captures](#) collections:

| Pattern | Description |
|------------------|--------------------------------------|
| <code>\b</code> | Begin the match at a word boundary. |
| <code>\w+</code> | Matches one or more word characters. |

| Pattern | Description |
|------------------------|--|
| [;,]? | Matches zero or one comma or semicolon. |
| \s? | Matches zero or one white-space character. |
| (\w+ [;,]? \s?)+ | Matches one or more occurrences of one or more word characters followed by an optional comma or semicolon followed by an optional white-space character. This pattern defines the first capturing group, which is necessary so that the combination of multiple word characters (that is, a word) followed by an optional punctuation symbol will be repeated until the regular expression engine reaches the end of a sentence. |
| [.?!] | Matches a period, question mark, or exclamation point. |

As the following example shows, when a match is found, both the [GroupCollection](#) and [CaptureCollection](#) objects are populated with captures from the match. In this case, the capturing group `(\w+[,]?[\s?])` exists so that the `+` quantifier can be applied to it, which enables the regular expression pattern to match each word in a sentence. Otherwise, it would match the last word in a sentence.

C#

```
using System;
using System.Text.RegularExpressions;

public class Example
{
    public static void Main()
    {
        string input = "This is one sentence. This is another.";
        string pattern = @"\b(\w+[,]?[\s?])+[.?!]";

        foreach (Match match in Regex.Matches(input, pattern)) {
            Console.WriteLine("Match: '{0}' at index {1}.",
                match.Value, match.Index);
            int grpCtr = 0;
            foreach (Group grp in match.Groups) {
                Console.WriteLine("    Group {0}: '{1}' at index {2}.",
                    grpCtr, grp.Value, grp.Index);
                int capCtr = 0;
                foreach (Capture cap in grp.Captures) {
                    Console.WriteLine("        Capture {0}: '{1}' at {2}.",
                        capCtr, cap.Value, cap.Index);
                    capCtr++;
                }
                grpCtr++;
            }
            Console.WriteLine();
        }
    }
}
```

```

}

// The example displays the following output:
//      Match: 'This is one sentence.' at index 0.
//          Group 0: 'This is one sentence.' at index 0.
//              Capture 0: 'This is one sentence.' at 0.
//          Group 1: 'sentence' at index 12.
//              Capture 0: 'This ' at 0.
//              Capture 1: 'is ' at 5.
//              Capture 2: 'one ' at 8.
//              Capture 3: 'sentence' at 12.
//
//      Match: 'This is another.' at index 22.
//          Group 0: 'This is another.' at index 22.
//              Capture 0: 'This is another.' at 22.
//          Group 1: 'another' at index 30.
//              Capture 0: 'This ' at 22.
//              Capture 1: 'is ' at 27.
//              Capture 2: 'another' at 30.

```

When you use subexpressions only to apply quantifiers to them and you aren't interested in the captured text, you should disable group captures. For example, the `(?:subexpression)` language element prevents the group to which it applies from capturing matched substrings. In the following example, the regular expression pattern from the previous example is changed to `\b(?:\w+[,]?\s?)+[.?!]`. As the output shows, it prevents the regular expression engine from populating the [GroupCollection](#) and [CaptureCollection](#) collections:

C#

```

using System;
using System.Text.RegularExpressions;

public class Example
{
    public static void Main()
    {
        string input = "This is one sentence. This is another.";
        string pattern = @"\b(?:\w+[, ]?\s?)+[.?!]";

        foreach (Match match in Regex.Matches(input, pattern)) {
            Console.WriteLine("Match: '{0}' at index {1}.",
                match.Value, match.Index);
            int grpCtr = 0;
            foreach (Group grp in match.Groups) {
                Console.WriteLine("    Group {0}: '{1}' at index {2}.",
                    grpCtr, grp.Value, grp.Index);
                int capCtr = 0;
                foreach (Capture cap in grp.Captures) {
                    Console.WriteLine("        Capture {0}: '{1}' at {2}.",
                        capCtr, cap.Value, cap.Index);
                    capCtr++;
                }
            }
        }
    }
}

```

```

        }
        grpCtr++;
    }
    Console.WriteLine();
}
}

// The example displays the following output:
//      Match: 'This is one sentence.' at index 0.
//          Group 0: 'This is one sentence.' at index 0.
//              Capture 0: 'This is one sentence.' at 0.
//
//      Match: 'This is another.' at index 22.
//          Group 0: 'This is another.' at index 22.
//              Capture 0: 'This is another.' at 22.

```

You can disable captures in one of the following ways:

- Use the `(?:subexpression)` language element. This element prevents the capture of matched substrings in the group to which it applies. It doesn't disable substring captures in any nested groups.
- Use the [ExplicitCapture](#) option. It disables all unnamed or implicit captures in the regular expression pattern. When you use this option, only substrings that match named groups defined with the `(?<name>subexpression)` language element can be captured. The [ExplicitCapture](#) flag can be passed to the `options` parameter of a [Regex](#) class constructor or to the `options` parameter of a [Regex](#) static matching method.
- Use the `n` option in the `(?imnsx)` language element. This option disables all unnamed or implicit captures from the point in the regular expression pattern at which the element appears. Captures are disabled either until the end of the pattern or until the `(-n)` option enables unnamed or implicit captures. For more information, see [Miscellaneous Constructs](#).
- Use the `n` option in the `(?imnsx:subexpression)` language element. This option disables all unnamed or implicit captures in `subexpression`. Captures by any unnamed or implicit nested capturing groups are disabled as well.

Related articles

| Title | Description |
|--|--|
| Details of Regular Expression Behavior | Examines the implementation of the regular expression engine in .NET. The article focuses on the flexibility of regular expressions and explains |

| Title | Description |
|---|--|
| | the developer's responsibility for ensuring the efficient and robust operation of the regular expression engine. |
| Backtracking | Explains what backtracking is and how it affects regular expression performance, and examines language elements that provide alternatives to backtracking. |
| Regular Expression Language - Quick Reference | Describes the elements of the regular expression language in .NET and provides links to detailed documentation for each language element. |

 **Collaborate with us on GitHub**

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

The Regular Expression Object Model

Article • 09/15/2021

This topic describes the object model used in working with .NET regular expressions. It contains the following sections:

- [The Regular Expression Engine](#)
- [The MatchCollection and Match Objects](#)
- [The Group Collection](#)
- [The Captured Group](#)
- [The Capture Collection](#)
- [The Individual Capture](#)

The Regular Expression Engine

The regular expression engine in .NET is represented by the [Regex](#) class. The regular expression engine is responsible for parsing and compiling a regular expression, and for performing operations that match the regular expression pattern with an input string. The engine is the central component in the .NET regular expression object model.

You can use the regular expression engine in either of two ways:

- By calling the static methods of the [Regex](#) class. The method parameters include the input string and the regular expression pattern. The regular expression engine caches regular expressions that are used in static method calls, so repeated calls to static regular expression methods that use the same regular expression offer relatively good performance.
- By instantiating a [Regex](#) object, by passing a regular expression to the class constructor. In this case, the [Regex](#) object is immutable (read-only) and represents a regular expression engine that is tightly coupled with a single regular expression. Because regular expressions used by [Regex](#) instances are not cached, you should not instantiate a [Regex](#) object multiple times with the same regular expression.

You can call the methods of the [Regex](#) class to perform the following operations:

- Determine whether a string matches a regular expression pattern.
- Extract a single match or the first match.

- Extract all matches.
- Replace a matched substring.
- Split a single string into an array of strings.

These operations are described in the following sections.

Matching a Regular Expression Pattern

The `Regex.IsMatch` method returns `true` if the string matches the pattern, or `false` if it does not. The `IsMatch` method is often used to validate string input. For example, the following code ensures that a string matches a valid social security number in the United States.

C#

```
using System;
using System.Text.RegularExpressions;

public class Example
{
    public static void Main()
    {
        string[] values = { "111-22-3333", "111-2-3333" };
        string pattern = @"^\d{3}-\d{2}-\d{4}$";
        foreach (string value in values) {
            if (Regex.IsMatch(value, pattern))
                Console.WriteLine("{0} is a valid SSN.", value);
            else
                Console.WriteLine("{0}: Invalid", value);
        }
    }
}
// The example displays the following output:
//      111-22-3333 is a valid SSN.
//      111-2-3333: Invalid
```

The regular expression pattern `^\d{3}-\d{2}-\d{4}$` is interpreted as shown in the following table.

| Pattern | Description |
|--------------------|--|
| <code>^</code> | Match the beginning of the input string. |
| <code>\d{3}</code> | Match three decimal digits. |
| <code>-</code> | Match a hyphen. |

| Pattern | Description |
|---------|------------------------------------|
| \d{2} | Match two decimal digits. |
| - | Match a hyphen. |
| \d{4} | Match four decimal digits. |
| \$ | Match the end of the input string. |

Extracting a Single Match or the First Match

The [Regex.Match](#) method returns a [Match](#) object that contains information about the first substring that matches a regular expression pattern. If the [Match.Success](#) property returns `true`, indicating that a match was found, you can retrieve information about subsequent matches by calling the [Match.NextMatch](#) method. These method calls can continue until the [Match.Success](#) property returns `false`. For example, the following code uses the [Regex.Match\(String, String\)](#) method to find the first occurrence of a duplicated word in a string. It then calls the [Match.NextMatch](#) method to find any additional occurrences. The example examines the [Match.Success](#) property after each method call to determine whether the current match was successful and whether a call to the [Match.NextMatch](#) method should follow.

C#

```

using System;
using System.Text.RegularExpressions;

public class Example
{
    public static void Main()
    {
        string input = "This is a a farm that that raises dairy cattle.";
        string pattern = @"\b(\w+)\W+(\1)\b";
        Match match = Regex.Match(input, pattern);
        while (match.Success)
        {
            Console.WriteLine("Duplicate '{0}' found at position {1}.",
                match.Groups[1].Value, match.Groups[2].Index);
            match = match.NextMatch();
        }
    }
}
// The example displays the following output:
//      Duplicate 'a' found at position 10.
//      Duplicate 'that' found at position 22.

```

The regular expression pattern `\b(\w+)\W+(\1)\b` is interpreted as shown in the following table.

| Pattern | Description |
|--------------------|---|
| <code>\b</code> | Begin the match on a word boundary. |
| <code>(\w+)</code> | Match one or more word characters. This is the first capturing group. |
| <code>\W+</code> | Match one or more non-word characters. |
| <code>(\1)</code> | Match the first captured string. This is the second capturing group. |
| <code>\b</code> | End the match on a word boundary. |

Extracting All Matches

The [Regex.Matches](#) method returns a [MatchCollection](#) object that contains information about all matches that the regular expression engine found in the input string. For example, the previous example could be rewritten to call the [Matches](#) method instead of the [Match](#) and [NextMatch](#) methods.

C#

```
using System;
using System.Text.RegularExpressions;

public class Example
{
    public static void Main()
    {
        string input = "This is a a farm that that raises dairy cattle.";
        string pattern = @"\b(\w+)\W+(\1)\b";
        foreach (Match match in Regex.Matches(input, pattern))
            Console.WriteLine("Duplicate '{0}' found at position {1}.",
                match.Groups[1].Value, match.Groups[2].Index);
    }
}
// The example displays the following output:
//      Duplicate 'a' found at position 10.
//      Duplicate 'that' found at position 22.
```

Replacing a Matched Substring

The [Regex.Replace](#) method replaces each substring that matches the regular expression pattern with a specified string or regular expression pattern, and returns the entire input

string with replacements. For example, the following code adds a U.S. currency symbol before a decimal number in a string.

```
C#  
  
using System;  
using System.Text.RegularExpressions;  
  
public class Example  
{  
    public static void Main()  
    {  
        string pattern = @"\b\d+\.\d{2}\b";  
        string replacement = "$$$&";  
        string input = "Total Cost: 103.64";  
        Console.WriteLine(Regex.Replace(input, pattern, replacement));  
    }  
}  
// The example displays the following output:  
//      Total Cost: $103.64
```

The regular expression pattern `\b\d+\.\d{2}\b` is interpreted as shown in the following table.

| Pattern | Description |
|--------------------|-------------------------------------|
| <code>\b</code> | Begin the match at a word boundary. |
| <code>\d+</code> | Match one or more decimal digits. |
| <code>\.</code> | Match a period. |
| <code>\d{2}</code> | Match two decimal digits. |
| <code>\b</code> | End the match at a word boundary. |

The replacement pattern `$$$&` is interpreted as shown in the following table.

| Pattern | Replacement string |
|----------------------|---------------------------------|
| <code>\$\$</code> | The dollar sign (\$) character. |
| <code>\$&</code> | The entire matched substring. |

Splitting a Single String into an Array of Strings

The `Regex.Split` method splits the input string at the positions defined by a regular expression match. For example, the following code places the items in a numbered list

into a string array.

C#

```
using System;
using System.Text.RegularExpressions;

public class Example
{
    public static void Main()
    {
        string input = "1. Eggs 2. Bread 3. Milk 4. Coffee 5. Tea";
        string pattern = @"\b\d{1,2}\.\s";
        foreach (string item in Regex.Split(input, pattern))
        {
            if (!String.IsNullOrEmpty(item))
                Console.WriteLine(item);
        }
    }
}
// The example displays the following output:
//      Eggs
//      Bread
//      Milk
//      Coffee
//      Tea
```

The regular expression pattern `\b\d{1,2}\.\s` is interpreted as shown in the following table.

| Pattern | Description |
|----------------------|-------------------------------------|
| <code>\b</code> | Begin the match at a word boundary. |
| <code>\d{1,2}</code> | Match one or two decimal digits. |
| <code>\.</code> | Match a period. |
| <code>\s</code> | Match a white-space character. |

The MatchCollection and Match Objects

Regex methods return two objects that are part of the regular expression object model: the [MatchCollection](#) object, and the [Match](#) object.

The Match Collection

The `Regex.Matches` method returns a `MatchCollection` object that contains `Match` objects that represent all the matches that the regular expression engine found, in the order in which they occur in the input string. If there are no matches, the method returns a `MatchCollection` object with no members. The `MatchCollection.Item[]` property lets you access individual members of the collection by index, from zero to one less than the value of the `MatchCollection.Count` property. `Item[]` is the collection's indexer (in C#) and default property (in Visual Basic).

By default, the call to the `Regex.Matches` method uses lazy evaluation to populate the `MatchCollection` object. Access to properties that require a fully populated collection, such as the `MatchCollection.Count` and `MatchCollection.Item[]` properties, may involve a performance penalty. As a result, we recommend that you access the collection by using the `IEnumerator` object that is returned by the `MatchCollection.GetEnumerator` method. Individual languages provide constructs, such as `For Each` in Visual Basic and `foreach` in C#, that wrap the collection's `IEnumerator` interface.

The following example uses the `Regex.Matches(String)` method to populate a `MatchCollection` object with all the matches found in an input string. The example enumerates the collection, copies the matches to a string array, and records the character positions in an integer array.

C#

```
using System;
using System.Collections.Generic;
using System.Text.RegularExpressions;

public class Example
{
    public static void Main()
    {
        MatchCollection matches;
        List<string> results = new List<string>();
        List<int> matchposition = new List<int>();

        // Create a new Regex object and define the regular expression.
        Regex r = new Regex("abc");
        // Use the Matches method to find all matches in the input string.
        matches = r.Matches("123abc4abcd");
        // Enumerate the collection to retrieve all matches and positions.
        foreach (Match match in matches)
        {
            // Add the match string to the string array.
            results.Add(match.Value);
            // Record the character position where the match was found.
            matchposition.Add(match.Index);
        }
        // List the results.
    }
}
```

```

        for (int ctr = 0; ctr < results.Count; ctr++)
            Console.WriteLine("{0}' found at position {1}.",
                               results[ctr], matchposition[ctr]);
    }
}
// The example displays the following output:
//      'abc' found at position 3.
//      'abc' found at position 7.

```

The Match

The [Match](#) class represents the result of a single regular expression match. You can access [Match](#) objects in two ways:

- By retrieving them from the [MatchCollection](#) object that is returned by the [Regex.Matches](#) method. To retrieve individual [Match](#) objects, iterate the collection by using a `foreach` (in C#) or `For Each...Next` (in Visual Basic) construct, or use the [MatchCollection.Item\[\]](#) property to retrieve a specific [Match](#) object either by index or by name. You can also retrieve individual [Match](#) objects from the collection by iterating the collection by index, from zero to one less than the number of objects in the collection. However, this method does not take advantage of lazy evaluation, because it accesses the [MatchCollection.Count](#) property.

The following example retrieves individual [Match](#) objects from a [MatchCollection](#) object by iterating the collection using the `foreach` OR `For Each...Next` construct. The regular expression simply matches the string "abc" in the input string.

C#

```

using System;
using System.Text.RegularExpressions;

public class Example
{
    public static void Main()
    {
        string pattern = "abc";
        string input = "abc123abc456abc789";
        foreach (Match match in Regex.Matches(input, pattern))
            Console.WriteLine("{0} found at position {1}.",
                               match.Value, match.Index);
    }
}
// The example displays the following output:
//      abc found at position 0.
//      abc found at position 6.
//      abc found at position 12.

```

- By calling the `Regex.Match` method, which returns a `Match` object that represents the first match in a string or a portion of a string. You can determine whether the match has been found by retrieving the value of the `Match.Success` property. To retrieve `Match` objects that represent subsequent matches, call the `Match.NextMatch` method repeatedly, until the `Success` property of the returned `Match` object is `false`.

The following example uses the `Regex.Match(String, String)` and `Match.NextMatch` methods to match the string "abc" in the input string.

```
C#  
  
using System;  
using System.Text.RegularExpressions;  
  
public class Example  
{  
    public static void Main()  
    {  
        string pattern = "abc";  
        string input = "abc123abc456abc789";  
        Match match = Regex.Match(input, pattern);  
        while (match.Success)  
        {  
            Console.WriteLine("{0} found at position {1}.",  
                match.Value, match.Index);  
            match = match.NextMatch();  
        }  
    }  
}  
// The example displays the following output:  
//      abc found at position 0.  
//      abc found at position 6.  
//      abc found at position 12.
```

Two properties of the `Match` class return collection objects:

- The `Match.Groups` property returns a `GroupCollection` object that contains information about the substrings that match capturing groups in the regular expression pattern.
- The `Match.Captures` property returns a `CaptureCollection` object that is of limited use. The collection is not populated for a `Match` object whose `Success` property is `false`. Otherwise, it contains a single `Capture` object that has the same information as the `Match` object.

For more information about these objects, see [The Group Collection](#) and [The Capture Collection](#) sections later in this topic.

Two additional properties of the [Match](#) class provide information about the match. The `Match.Value` property returns the substring in the input string that matches the regular expression pattern. The `Match.Index` property returns the zero-based starting position of the matched string in the input string.

The [Match](#) class also has two pattern-matching methods:

- The [Match.NextMatch](#) method finds the match after the match represented by the current [Match](#) object, and returns a [Match](#) object that represents that match.
- The [Match.Result](#) method performs a specified replacement operation on the matched string and returns the result.

The following example uses the [Match.Result](#) method to prepend a \$ symbol and a space before every number that includes two fractional digits.

C#

```
using System;
using System.Text.RegularExpressions;

public class Example
{
    public static void Main()
    {
        string pattern = @"\b\d+(,\d{3})*\.\d{2}\b";
        string input = "16.32\n194.03\n1,903,672.08";

        foreach (Match match in Regex.Matches(input, pattern))
            Console.WriteLine(match.Result("$$ $&"));
    }
}
// The example displays the following output:
//      $ 16.32
//      $ 194.03
//      $ 1,903,672.08
```

The regular expression pattern `\b\d+(,\d{3})*\.\d{2}\b` is defined as shown in the following table.

| Pattern | Description |
|------------------|-------------------------------------|
| <code>\b</code> | Begin the match at a word boundary. |
| <code>\d+</code> | Match one or more decimal digits. |

| Pattern | Description |
|-----------|---|
| (,\d{3})* | Match zero or more occurrences of a comma followed by three decimal digits. |
| \. | Match the decimal point character. |
| \d{2} | Match two decimal digits. |
| \b | End the match at a word boundary. |

The replacement pattern `$$ $&` indicates that the matched substring should be replaced by a dollar sign (\$) symbol (the `$$` pattern), a space, and the value of the match (the `$&` pattern).

[Back to top](#)

The Group Collection

The `Match.Groups` property returns a `GroupCollection` object that contains `Group` objects that represent captured groups in a single match. The first `Group` object in the collection (at index 0) represents the entire match. Each object that follows represents the results of a single capturing group.

You can retrieve individual `Group` objects in the collection by using the `GroupCollection.Item[]` property. You can retrieve unnamed groups by their ordinal position in the collection, and retrieve named groups either by name or by ordinal position. Unnamed captures appear first in the collection, and are indexed from left to right in the order in which they appear in the regular expression pattern. Named captures are indexed after unnamed captures, from left to right in the order in which they appear in the regular expression pattern. To determine what numbered groups are available in the collection returned for a particular regular expression matching method, you can call the instance `Regex.GetGroupNumbers` method. To determine what named groups are available in the collection, you can call the instance `Regex.GetGroupNames` method. Both methods are particularly useful in general-purpose routines that analyze the matches found by any regular expression.

The `GroupCollection.Item[]` property is the indexer of the collection in C# and the collection object's default property in Visual Basic. This means that individual `Group` objects can be accessed by index (or by name, in the case of named groups) as follows:

C#

```
Group group = match.Groups[ctr];
```

The following example defines a regular expression that uses grouping constructs to capture the month, day, and year of a date.

```
C#  
  
using System;  
using System.Text.RegularExpressions;  
  
public class Example  
{  
    public static void Main()  
    {  
        string pattern = @"\b(\w+)\s(\d{1,2}),\s(\d{4})\b";  
        string input = "Born: July 28, 1989";  
        Match match = Regex.Match(input, pattern);  
        if (match.Success)  
            for (int ctr = 0; ctr < match.Groups.Count; ctr++)  
                Console.WriteLine("Group {0}: {1}", ctr,  
match.Groups[ctr].Value);  
    }  
}  
// The example displays the following output:  
//      Group 0: July 28, 1989  
//      Group 1: July  
//      Group 2: 28  
//      Group 3: 1989
```

The regular expression pattern `\b(\w+)\s(\d{1,2}),\s(\d{4})\b` is defined as shown in the following table.

| Pattern | Description |
|------------------------|---|
| <code>\b</code> | Begin the match at a word boundary. |
| <code>(\w+)</code> | Match one or more word characters. This is the first capturing group. |
| <code>\s</code> | Match a white-space character. |
| <code>(\d{1,2})</code> | Match one or two decimal digits. This is the second capturing group. |
| <code>,</code> | Match a comma. |
| <code>\s</code> | Match a white-space character. |
| <code>(\d{4})</code> | Match four decimal digits. This is the third capturing group. |
| <code>\b</code> | End the match on a word boundary. |

[Back to top](#)

The Captured Group

The [Group](#) class represents the result from a single capturing group. Group objects that represent the capturing groups defined in a regular expression are returned by the [Item\[\]](#) property of the [GroupCollection](#) object returned by the [Match.Groups](#) property. The [Item\[\]](#) property is the indexer (in C#) and the default property (in Visual Basic) of the [Group](#) class. You can also retrieve individual members by iterating the collection using the `foreach` or `For Each` construct. For an example, see the previous section.

The following example uses nested grouping constructs to capture substrings into groups. The regular expression pattern `(a(b))c` matches the string "abc". It assigns the substring "ab" to the first capturing group, and the substring "b" to the second capturing group.

C#

```
var matchposition = new List<int>();
var results = new List<string>();
// Define substrings abc, ab, b.
var r = new Regex("(a(b))c");
Match m = r.Match("abdabc");
for (int i = 0; m.Groups[i].Value != ""; i++)
{
    // Add groups to string array.
    results.Add(m.Groups[i].Value);
    // Record character position.
    matchposition.Add(m.Groups[i].Index);
}

// Display the capture groups.
for (int ctr = 0; ctr < results.Count; ctr++)
    Console.WriteLine("{0} at position {1}",
                      results[ctr], matchposition[ctr]);
// The example displays the following output:
//      abc at position 3
//      ab at position 3
//      b at position 4
```

The following example uses named grouping constructs to capture substrings from a string that contains data in the format "DATANAME:VALUE", which the regular expression splits at the colon (:).

C#

```
var r = new Regex(@"^(?<name>\w+):(?:<value>\w+)");
Match m = r.Match("Section1:119900");
Console.WriteLine(m.Groups["name"].Value);
Console.WriteLine(m.Groups["value"].Value);
```

```
// The example displays the following output:  
//      Section1  
//      119900
```

The regular expression pattern `^(?<name>\w+):(?<value>\w+)` is defined as shown in the following table.

| Pattern | Description |
|----------------------------------|---|
| <code>^</code> | Begin the match at the beginning of the input string. |
| <code>(?<name>\w+)</code> | Match one or more word characters. The name of this capturing group is <code>name</code> . |
| <code>:</code> | Match a colon. |
| <code>(?<value>\w+)</code> | Match one or more word characters. The name of this capturing group is <code>value</code> . |

The properties of the [Group](#) class provide information about the captured group: The `Group.Value` property contains the captured substring, the `Group.Index` property indicates the starting position of the captured group in the input text, the `Group.Length` property contains the length of the captured text, and the `Group.Success` property indicates whether a substring matched the pattern defined by the capturing group.

Applying quantifiers to a group (for more information, see [Quantifiers](#)) modifies the relationship of one capture per capturing group in two ways:

- If the `*` or `*?` quantifier (which specifies zero or more matches) is applied to a group, a capturing group may not have a match in the input string. When there is no captured text, the properties of the [Group](#) object are set as shown in the following table.

| Group property | Value |
|----------------------|---------------------------|
| <code>Success</code> | <code>false</code> |
| <code>Value</code> | <code>String.Empty</code> |
| <code>Length</code> | 0 |

The following example provides an illustration. In the regular expression pattern `aaa(bbb)*ccc`, the first capturing group (the substring "bbb") can be matched zero or more times. Because the input string "aaaccc" matches the pattern, the capturing group does not have a match.

C#

```

using System;
using System.Text.RegularExpressions;

public class Example
{
    public static void Main()
    {
        string pattern = "aaa(bbb)*ccc";
        string input = "aaaccc";
        Match match = Regex.Match(input, pattern);
        Console.WriteLine("Match value: {0}", match.Value);
        if (match.Groups[1].Success)
            Console.WriteLine("Group 1 value: {0}",
        match.Groups[1].Value);
        else
            Console.WriteLine("The first capturing group has no match.");
    }
}
// The example displays the following output:
//      Match value: aaaccc
//      The first capturing group has no match.

```

- Quantifiers can match multiple occurrences of a pattern that is defined by a capturing group. In this case, the `Value` and `Length` properties of a `Group` object contain information only about the last captured substring. For example, the following regular expression matches a single sentence that ends in a period. It uses two grouping constructs: The first captures individual words along with a white-space character; the second captures individual words. As the output from the example shows, although the regular expression succeeds in capturing an entire sentence, the second capturing group captures only the last word.

C#

```

using System;
using System.Text.RegularExpressions;

public class Example
{
    public static void Main()
    {
        string pattern = @"\b((\w+)\s?)+\.?";
        string input = "This is a sentence. This is another sentence.";
        Match match = Regex.Match(input, pattern);
        if (match.Success)
        {
            Console.WriteLine("Match: " + match.Value);
            Console.WriteLine("Group 2: " + match.Groups[2].Value);
        }
    }
}

```

```
// The example displays the following output:  
//      Match: This is a sentence.  
//      Group 2: sentence
```

[Back to top](#)

The Capture Collection

The [Group](#) object contains information only about the last capture. However, the entire set of captures made by a capturing group is still available from the [CaptureCollection](#) object that is returned by the [Group.Captures](#) property. Each member of the collection is a [Capture](#) object that represents a capture made by that capturing group, in the order in which they were captured (and, therefore, in the order in which the captured strings were matched from left to right in the input string). You can retrieve individual [Capture](#) objects from the collection in either of two ways:

- By iterating through the collection using a construct such as `foreach` (in C#) or `For Each` (in Visual Basic).
- By using the [CaptureCollection.Item\[\]](#) property to retrieve a specific object by index. The [Item\[\]](#) property is the [CaptureCollection](#) object's default property (in Visual Basic) or indexer (in C#).

If a quantifier is not applied to a capturing group, the [CaptureCollection](#) object contains a single [Capture](#) object that is of little interest, because it provides information about the same match as its [Group](#) object. If a quantifier is applied to a capturing group, the [CaptureCollection](#) object contains all captures made by the capturing group, and the last member of the collection represents the same capture as the [Group](#) object.

For example, if you use the regular expression pattern `((a(b))c)+` (where the `+` quantifier specifies one or more matches) to capture matches from the string "abcabcabc", the [CaptureCollection](#) object for each [Group](#) object contains three members.

C#

```
using System;  
using System.Text.RegularExpressions;  
  
public class Example  
{  
    public static void Main()  
    {  
        string pattern = "((a(b))c)+";  
        string input = "abcabcabc";
```

```

Match match = Regex.Match(input, pattern);
if (match.Success)
{
    Console.WriteLine("Match: '{0}' at position {1}",
                      match.Value, match.Index);
    GroupCollection groups = match.Groups;
    for (int ctr = 0; ctr < groups.Count; ctr++) {
        Console.WriteLine("    Group {0}: '{1}' at position {2}",
                          ctr, groups[ctr].Value, groups[ctr].Index);
        CaptureCollection captures = groups[ctr].Captures;
        for (int ctr2 = 0; ctr2 < captures.Count; ctr2++) {
            Console.WriteLine("        Capture {0}: '{1}' at position {2}",
                              ctr2, captures[ctr2].Value,
                              captures[ctr2].Index);
        }
    }
}
// The example displays the following output:
//     Match: 'abcababc' at position 0
//         Group 0: 'abcababc' at position 0
//             Capture 0: 'abcababc' at position 0
//             Group 1: 'abc' at position 6
//                 Capture 0: 'abc' at position 0
//                 Capture 1: 'abc' at position 3
//                 Capture 2: 'abc' at position 6
//             Group 2: 'ab' at position 6
//                 Capture 0: 'ab' at position 0
//                 Capture 1: 'ab' at position 3
//                 Capture 2: 'ab' at position 6
//             Group 3: 'b' at position 7
//                 Capture 0: 'b' at position 1
//                 Capture 1: 'b' at position 4
//                 Capture 2: 'b' at position 7

```

The following example uses the regular expression `(Abc)+` to find one or more consecutive runs of the string "Abc" in the string "XYZAbcAbcAbcXYZAbcAb". The example illustrates the use of the `Group.Captures` property to return multiple groups of captured substrings.

C#

```

int counter;
Match m;
CaptureCollection cc;
GroupCollection gc;

// Look for groupings of "Abc".
var r = new Regex("(Abc)+");
// Define the string to search.

```

```

m = r.Match("XYZAbcAbcAbcXYZAbcAb");
gc = m.Groups;

// Display the number of groups.
Console.WriteLine("Captured groups = " + gc.Count.ToString());

// Loop through each group.
for (int i = 0; i < gc.Count; i++)
{
    cc = gc[i].Captures;
    counter = cc.Count;

    // Display the number of captures in this group.
    Console.WriteLine("Captures count = " + counter.ToString());

    // Loop through each capture in the group.
    for (int ii = 0; ii < counter; ii++)
    {
        // Display the capture and its position.
        Console.WriteLine(cc[ii] + " Starts at character " +
            cc[ii].Index);
    }
}

// The example displays the following output:
//     Captured groups = 2
//     Captures count = 1
//     AbcAbcAbc Starts at character 3
//     Captures count = 3
//     Abc Starts at character 3
//     Abc Starts at character 6
//     Abc Starts at character 9

```

[Back to top](#)

The Individual Capture

The [Capture](#) class contains the results from a single subexpression capture. The [Capture.Value](#) property contains the matched text, and the [Capture.Index](#) property indicates the zero-based position in the input string at which the matched substring begins.

The following example parses an input string for the temperature of selected cities. A comma (",") is used to separate a city and its temperature, and a semicolon (";") is used to separate each city's data. The entire input string represents a single match. In the regular expression pattern `((\w+(\s\w+)*),(\d+);)+`, which is used to parse the string, the city name is assigned to the second capturing group, and the temperature is assigned to the fourth capturing group.

C#

```
using System;
using System.Text.RegularExpressions;

public class Example
{
    public static void Main()
    {
        string input = "Miami,78;Chicago,62;New York,67;San
Francisco,59;Seattle,58;";
        string pattern = @"((\w+(\s\w+)*),(\d+);)+";
        Match match = Regex.Match(input, pattern);
        if (match.Success)
        {
            Console.WriteLine("Current temperatures:");
            for (int ctr = 0; ctr < match.Groups[2].Captures.Count; ctr++)
                Console.WriteLine("{0,-20} {1,3}",
match.Groups[2].Captures[ctr].Value,
                                match.Groups[4].Captures[ctr].Value);
        }
    }
}
// The example displays the following output:
//      Current temperatures:
//      Miami          78
//      Chicago        62
//      New York       67
//      San Francisco  59
//      Seattle         58
```

The regular expression is defined as shown in the following table.

| Pattern | Description |
|----------------|---|
| \w+ | Match one or more word characters. |
| (\s\w+)* | Match zero or more occurrences of a white-space character followed by one or more word characters. This pattern matches multi-word city names. This is the third capturing group. |
| (\w+(\s\w+)*) | Match one or more word characters followed by zero or more occurrences of a white-space character and one or more word characters. This is the second capturing group. |
| , | Match a comma. |
| (\d+) | Match one or more digits. This is the fourth capturing group. |
| ; | Match a semicolon. |

| Pattern | Description |
|-----------------------------|--|
| ((\w+(\s\w+)*), (\d+);)+ | Match the pattern of a word followed by any additional words followed by a comma, one or more digits, and a semicolon, one or more times. This is the first capturing group. |

See also

- [System.Text.RegularExpressions](#)
- [.NET Regular Expressions](#)
- [Regular Expression Language - Quick Reference](#)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

Details of regular expression behavior

Article • 09/15/2021

The .NET regular expression engine is a backtracking regular expression matcher that incorporates a traditional Nondeterministic Finite Automaton (NFA) engine such as that used by Perl, Python, Emacs, and Tcl. This distinguishes it from faster, but more limited, pure regular expression Deterministic Finite Automaton (DFA) engines such as those found in awk, egrep, or lex. This also distinguishes it from standardized, but slower, POSIX NFAs. The following section describes the three types of regular expression engines, and explains why regular expressions in .NET are implemented by using a traditional NFA engine.

Benefits of the NFA engine

When DFA engines perform pattern matching, their processing order is driven by the input string. The engine begins at the beginning of the input string and proceeds sequentially to determine whether the next character matches the regular expression pattern. They can guarantee to match the longest string possible. Because they never test the same character twice, DFA engines do not support backtracking. However, because a DFA engine contains only finite state, it cannot match a pattern with backreferences, and because it does not construct an explicit expansion, it cannot capture subexpressions.

Unlike DFA engines, when traditional NFA engines perform pattern matching, their processing order is driven by the regular expression pattern. As it processes a particular language element, the engine uses greedy matching; that is, it matches as much of the input string as it possibly can. But it also saves its state after successfully matching a subexpression. If a match eventually fails, the engine can return to a saved state so it can try additional matches. This process of abandoning a successful subexpression match so that later language elements in the regular expression can also match is known as *backtracking*. NFA engines use backtracking to test all possible expansions of a regular expression in a specific order and accept the first match. Because a traditional NFA engine constructs a specific expansion of the regular expression for a successful match, it can capture subexpression matches and matching backreferences. However, because a traditional NFA backtracks, it can visit the same state multiple times if it arrives at the state over different paths. As a result, it can run exponentially slowly in the worst case. Because a traditional NFA engine accepts the first match it finds, it can also leave other (possibly longer) matches undiscovered.

POSIX NFA engines are like traditional NFA engines, except that they continue to backtrack until they can guarantee that they have found the longest match possible. As a result, a POSIX NFA engine is slower than a traditional NFA engine, and when you use a POSIX NFA engine, you cannot favor a shorter match over a longer one by changing the order of the backtracking search.

Traditional NFA engines are favored by programmers because they offer greater control over string matching than either DFA or POSIX NFA engines. Although, in the worst case, they can run slowly, you can steer them to find matches in linear or polynomial time by using patterns that reduce ambiguities and limit backtracking. In other words, although NFA engines trade performance for power and flexibility, in most cases they offer good to acceptable performance if a regular expression is well written and avoids cases in which backtracking degrades performance exponentially.

ⓘ Note

For information about the performance penalty caused by excessive backtracking and ways to craft a regular expression to work around them, see [Backtracking](#).

.NET engine capabilities

To take advantage of the benefits of a traditional NFA engine, the .NET regular expression engine includes a complete set of constructs to enable programmers to steer the backtracking engine. These constructs can be used to find matches faster or to favor specific expansions over others.

Other features of the .NET regular expression engine include the following:

- Lazy quantifiers: `??`, `*?`, `+?`, `{n,m}?`. These constructs tell the backtracking engine to search the minimum number of repetitions first. In contrast, ordinary greedy quantifiers try to match the maximum number of repetitions first. The following example illustrates the difference between the two. A regular expression matches a sentence that ends in a number, and a capturing group is intended to extract that number. The regular expression `.+(\d+)\.` includes the greedy quantifier `.+`, which causes the regular expression engine to capture only the last digit of the number. In contrast, the regular expression `.+?(\d+)\.` includes the lazy quantifier `.+?`, which causes the regular expression engine to capture the entire number.

C#

```

using System;
using System.Text.RegularExpressions;

public class Example
{
    public static void Main()
    {
        string greedyPattern = @"^.+(\d+)\.";
        string lazyPattern = @"^.?(\d+)\.";
        string input = "This sentence ends with the number 107325.";
        Match match;

        // Match using greedy quantifier .+.
        match = Regex.Match(input, greedyPattern);
        if (match.Success)
            Console.WriteLine("Number at end of sentence (greedy): {0}",
{0},
                           match.Groups[1].Value);
        else
            Console.WriteLine("{0} finds no match.", greedyPattern);

        // Match using lazy quantifier .+?.
        match = Regex.Match(input, lazyPattern);
        if (match.Success)
            Console.WriteLine("Number at end of sentence (lazy): {0}",
                           match.Groups[1].Value);
        else
            Console.WriteLine("{0} finds no match.", lazyPattern);
    }
}

// The example displays the following output:
//      Number at end of sentence (greedy): 5
//      Number at end of sentence (lazy): 107325

```

The greedy and lazy versions of this regular expression are defined as shown in the following table:

| Pattern | Description |
|------------------------|--|
| .+ (greedy quantifier) | Match at least one occurrence of any character. This causes the regular expression engine to match the entire string, and then to backtrack as needed to match the remainder of the pattern. |
| .+? (lazy quantifier) | Match at least one occurrence of any character, but match as few as possible. |
| (\d+) | Match at least one numeric character, and assign it to the first capturing group. |
| \. | Match a period. |

For more information about lazy quantifiers, see [Quantifiers](#).

- Positive lookahead: `(?=subexpression)`. This feature allows the backtracking engine to return to the same spot in the text after matching a subexpression. It is useful for searching throughout the text by verifying multiple patterns that start from the same position. It also allows the engine to verify that a substring exists at the end of the match without including the substring in the matched text. The following example uses positive lookahead to extract the words in a sentence that are not followed by punctuation symbols.

C#

```
using System;
using System.Text.RegularExpressions;

public class Example
{
    public static void Main()
    {
        string pattern = @"\b[A-Z]+\b(?=\P{P})";
        string input = "If so, what comes next?";
        foreach (Match match in Regex.Matches(input, pattern,
RegexOptions.IgnoreCase))
            Console.WriteLine(match.Value);
    }
}
// The example displays the following output:
//      If
//      what
//      comes
```

The regular expression `\b[A-Z]+\b(?=\P{P})` is defined as shown in the following table.

| Pattern | Description |
|--|---|
| <code>\b</code> | Begin the match at a word boundary. |
| <code>[A-Z]+</code> | Match any alphabetic character one or more times. Because the <code>Regex.Matches</code> method is called with the <code>RegexOptions.IgnoreCase</code> option, the comparison is case-insensitive. |
| <code>\b</code> | End the match at a word boundary. |
| <code>(?=</code> <code>=\P{P})</code> | Look ahead to determine whether the next character is a punctuation symbol. If it is not, the match succeeds. |

For more information about positive lookahead assertions, see [Grouping Constructs](#).

- Negative lookahead: `(?! subexpression)`. This feature adds the ability to match an expression only if a subexpression fails to match. This is powerful for pruning a search, because it is often simpler to provide an expression for a case that should be eliminated than an expression for cases that must be included. For example, it is difficult to write an expression for words that do not begin with "non". The following example uses negative lookahead to exclude them.

C#

```
using System;
using System.Text.RegularExpressions;

public class Example
{
    public static void Main()
    {
        string pattern = @"\b(?!non)\w+\b";
        string input = "Nonsense is not always non-functional.";
        foreach (Match match in Regex.Matches(input, pattern,
RegexOptions.IgnoreCase))
            Console.WriteLine(match.Value);
    }
}
// The example displays the following output:
//      is
//      not
//      always
//      functional
```

The regular expression pattern `\b(?!non)\w+\b` is defined as shown in the following table.

| Pattern | Description |
|----------------------|--|
| <code>\b</code> | Begin the match at a word boundary. |
| <code>(?!non)</code> | Look ahead to ensure that the current string does not begin with "non". If it does, the match fails. |
| <code>(\w+)</code> | Match one or more word characters. |
| <code>\b</code> | End the match at a word boundary. |

For more information about negative lookahead assertions, see [Grouping Constructs](#).

- Conditional evaluation: `(?(expression)yes|no)` and `(?(name)yes|no)`, where *expression* is a subexpression to match, *name* is the name of a capturing group, *yes* is the string to match if *expression* is matched or *name* is a valid, non-empty captured group, and *no* is the subexpression to match if *expression* is not matched or *name* is not a valid, non-empty captured group. This feature allows the engine to search by using more than one alternate pattern, depending on the result of a previous subexpression match or the result of a zero-width assertion. This allows a more powerful form of backreference that permits, for example, matching a subexpression based on whether a previous subexpression was matched. The regular expression in the following example matches paragraphs that are intended for both public and internal use. Paragraphs intended only for internal use begin with a `<PRIVATE>` tag. The regular expression pattern `^(?<Pvt>\<PRIVATE\>)\s)?(?(Pvt)((\w+\p{P})?\s+)|((\w+\p{P})?\s+))\r?$` uses conditional evaluation to assign the contents of paragraphs intended for public and for internal use to separate capturing groups. These paragraphs can then be handled differently.

C#

```

using System;
using System.Text.RegularExpressions;

public class Example
{
    public static void Main()
    {
        string input = "<PRIVATE> This is not for public consumption."
+ Environment.NewLine +
                    "But this is for public consumption." +
Environment.NewLine +
                    "<PRIVATE> Again, this is confidential.\n";
        string pattern = @"^(?<Pvt>\<PRIVATE\>)\s)?(?(Pvt)((\w+\p{P})?\s+)|((\w+\p{P})?\s+))\r?$";
        string publicDocument = null, privateDocument = null;

        foreach (Match match in Regex.Matches(input, pattern,
RegexOptions.Multiline))
        {
            if (match.Groups[1].Success)
            {
                privateDocument += match.Groups[1].Value + "\n";
            }
            else
            {
                publicDocument += match.Groups[3].Value + "\n";
                privateDocument += match.Groups[3].Value + "\n";
            }
        }

        Console.WriteLine("Private Document:");
    }
}

```

```

        Console.WriteLine(privateDocument);
        Console.WriteLine("Public Document:");
        Console.WriteLine(publicDocument);
    }
}

// The example displays the following output:
//  Private Document:
//  This is not for public consumption.
//  But this is for public consumption.
//  Again, this is confidential.
//
//  Public Document:
//  But this is for public consumption.

```

The regular expression pattern is defined as shown in the following table.

| Pattern | Description |
|---|--|
| <code>^</code> | Begin the match at the beginning of a line. |
| <code>(?<Pvt>\<PRIVATE\>\s?)</code> | Match zero or one occurrence of the string <code><PRIVATE></code> followed by a white-space character. Assign the match to a capturing group named <code>Pvt</code> . |
| <code>(?(Pvt)((\w+\p{P})?\s)+)</code> | If the <code>Pvt</code> capturing group exists, match one or more occurrences of one or more word characters followed by zero or one punctuation separator followed by a white-space character. Assign the substring to the first capturing group. |
| <code> ((\w+\p{P})?\s+))</code> | If the <code>Pvt</code> capturing group does not exist, match one or more occurrences of one or more word characters followed by zero or one punctuation separator followed by a white-space character. Assign the substring to the third capturing group. |
| <code>\r?\$</code> | Match the end of a line or the end of the string. |

For more information about conditional evaluation, see [Alternation Constructs](#).

- Balancing group definitions: `(?<name1 - name2> subexpression)`. This feature allows the regular expression engine to keep track of nested constructs such as parentheses or opening and closing brackets. For an example, see [Grouping Constructs](#).
- Atomic groups: `(?> subexpression)`. This feature allows the backtracking engine to guarantee that a subexpression matches only the first match found for that subexpression, as if the expression were running independent of its containing expression. If you do not use this construct, backtracking searches from the larger expression can change the behavior of a subexpression. For example, the regular

expression `(a+)\w` matches one or more "a" characters, along with a word character that follows the sequence of "a" characters, and assigns the sequence of "a" characters to the first capturing group. However, if the final character of the input string is also an "a", it is matched by the `\w` language element and is not included in the captured group.

```
C#  
  
using System;  
using System.Text.RegularExpressions;  
  
public class Example  
{  
    public static void Main()  
    {  
        string[] inputs = { "aaaaa", "aaaaab" };  
        string backtrackingPattern = @"(a+)\w";  
        Match match;  
  
        foreach (string input in inputs)  
        {  
            Console.WriteLine("Input: {0}", input);  
            match = Regex.Match(input, backtrackingPattern);  
            Console.WriteLine("  Pattern: {0}", backtrackingPattern);  
            if (match.Success)  
            {  
                Console.WriteLine("      Match: {0}", match.Value);  
                Console.WriteLine("      Group 1: {0}",  
match.Groups[1].Value);  
            }  
            else  
            {  
                Console.WriteLine("      Match failed.");  
            }  
        }  
        Console.WriteLine();  
    }  
}  
// The example displays the following output:  
//      Input: aaaaa  
//      Pattern: (a+)\w  
//      Match: aaaaa  
//      Group 1: aaaa  
//      Input: aaaaab  
//      Pattern: (a+)\w  
//      Match: aaaaab  
//      Group 1: aaaa
```

The regular expression `((?>a+))\w` prevents this behavior. Because all consecutive "a" characters are matched without backtracking, the first capturing group includes

all consecutive "a" characters. If the "a" characters are not followed by at least one more character other than "a", the match fails.

```
C#  
  
using System;  
using System.Text.RegularExpressions;  
  
public class Example  
{  
    public static void Main()  
    {  
        string[] inputs = { "aaaaa", "aaaaab" };  
        string nonbacktrackingPattern = @"((?>a+))\w";  
        Match match;  
  
        foreach (string input in inputs)  
        {  
            Console.WriteLine("Input: {0}", input);  
            match = Regex.Match(input, nonbacktrackingPattern);  
            Console.WriteLine("    Pattern: {0}",  
nonbacktrackingPattern);  
            if (match.Success)  
            {  
                Console.WriteLine("        Match: {0}", match.Value);  
                Console.WriteLine("        Group 1: {0}",  
match.Groups[1].Value);  
            }  
            else  
            {  
                Console.WriteLine("        Match failed.");  
            }  
        }  
        Console.WriteLine();  
    }  
}  
// The example displays the following output:  
//     Input: aaaaa  
//             Pattern: ((?>a+))\w  
//             Match failed.  
//     Input: aaaaab  
//             Pattern: ((?>a+))\w  
//             Match: aaaaab  
//             Group 1: aaaaa
```

For more information about atomic groups, see [Grouping Constructs](#).

- Right-to-left matching, which is specified by supplying the `RegexOptions.RightToLeft` option to a `Regex` class constructor or static instance matching method. This feature is useful when searching from right to left instead of from left to right, or in cases where it is more efficient to begin a match at the

right part of the pattern instead of the left. As the following example illustrates, using right-to-left matching can change the behavior of greedy quantifiers. The example conducts two searches for a sentence that ends in a number. The left-to-right search that uses the greedy quantifier `+ matches one of the six digits in the sentence, whereas the right-to-left search matches all six digits. For a description of the regular expression pattern, see the example that illustrates lazy quantifiers earlier in this section.`

C#

```
using System;
using System.Text.RegularExpressions;

public class Example
{
    public static void Main()
    {
        string greedyPattern = @"^.+(\d+)\.";
        string input = "This sentence ends with the number 107325.";
        Match match;

        // Match from left-to-right using lazy quantifier .+?.
        match = Regex.Match(input, greedyPattern);
        if (match.Success)
            Console.WriteLine("Number at end of sentence (left-to-right): {0}",
                               match.Groups[1].Value);
        else
            Console.WriteLine("{0} finds no match.", greedyPattern);

        // Match from right-to-left using greedy quantifier .+.
        match = Regex.Match(input, greedyPattern,
                           RegexOptions.RightToLeft);
        if (match.Success)
            Console.WriteLine("Number at end of sentence (right-to-left): {0}",
                               match.Groups[1].Value);
        else
            Console.WriteLine("{0} finds no match.", greedyPattern);
    }
}

// The example displays the following output:
//      Number at end of sentence (left-to-right): 5
//      Number at end of sentence (right-to-left): 107325
```

For more information about right-to-left matching, see [Regular Expression Options](#).

- Positive and negative lookbehind: `(?<= subexpression)` for positive lookbehind, and `(?<! subexpression)` for negative lookbehind. This feature is similar to

lookahead, which is discussed earlier in this topic. Because the regular expression engine allows complete right-to-left matching, regular expressions allow unrestricted lookbehinds. Positive and negative lookbehind can also be used to avoid nesting quantifiers when the nested subexpression is a superset of an outer expression. Regular expressions with such nested quantifiers often offer poor performance. For example, the following example verifies that a string begins and ends with an alphanumeric character, and that any other character in the string is one of a larger subset. It forms a portion of the regular expression used to validate email addresses; for more information, see [How to: Verify that Strings Are in Valid Email Format](#).

C#

```
using System;
using System.Text.RegularExpressions;

public class Example
{
    public static void Main()
    {
        string[] inputs = { "jack.sprat", "dog#", "dog#1", "me.myself",
                           "me.myself!" };
        string pattern = @"^([A-Z0-9]([-!#$%&'.*+/=?^`{}|~\w])*(<=[A-Z0-9])$";
        foreach (string input in inputs)
        {
            if (Regex.IsMatch(input, pattern, RegexOptions.IgnoreCase))
                Console.WriteLine("{0}: Valid", input);
            else
                Console.WriteLine("{0}: Invalid", input);
        }
    }
}
// The example displays the following output:
//      jack.sprat: Valid
//      dog#: Invalid
//      dog#1: Valid
//      me.myself: Valid
//      me.myself!: Invalid
```

The regular expression `^([A-Z0-9]([-!#$%&'.*+/=?^`{}|~\w])*(<=[A-Z0-9])$` is defined as shown in the following table.

| Pattern | Description |
|-----------------------|---|
| <code>^</code> | Begin the match at the beginning of the string. |
| <code>[A-Z0-9]</code> | Match any numeric or alphanumeric character. (The comparison is |

| Pattern | Description |
|---------------------------------|---|
| | case-insensitive.) |
| ([-!#\$%&'.*+/=? ^`{} ~\w])* | Match zero or more occurrences of any word character, or any of the following characters: -, !, #, \$, %, &, ', ., *, +, /, =, ?, ^, ` , {, }, , or ~. |
| (?<=[A-Z0-9]) | Look behind to the previous character, which must be numeric or alphanumeric. (The comparison is case-insensitive.) |
| \$ | End the match at the end of the string. |

For more information about positive and negative lookbehind, see [Grouping Constructs](#).

Related articles

| Title | Description |
|---|--|
| Backtracking | Provides information about how regular expression backtracking branches to find alternative matches. |
| Compilation and Reuse | Provides information about compiling and reusing regular expressions to increase performance. |
| Thread Safety | Provides information about regular expression thread safety and explains when you should synchronize access to regular expression objects. |
| .NET Regular Expressions | Provides an overview of the programming language aspect of regular expressions. |
| The Regular Expression Object Model | Provides information and code examples illustrating how to use the regular expression classes. |
| Regular Expression Language - Quick Reference | Provides information about the set of characters, operators, and constructs that you can use to define regular expressions. |

Reference

- [System.Text.RegularExpressions](#)

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

Backtracking in regular expressions

Article • 08/11/2023

Backtracking occurs when a regular expression pattern contains optional [quantifiers](#) or [alternation constructs](#), and the regular expression engine returns to a previous saved state to continue its search for a match. Backtracking is central to the power of regular expressions; it makes it possible for expressions to be powerful and flexible, and to match very complex patterns. At the same time, this power comes at a cost.

Backtracking is often the single most important factor that affects the performance of the regular expression engine. Fortunately, the developer has control over the behavior of the regular expression engine and how it uses backtracking. This topic explains how backtracking works and how it can be controlled.

⚠ Warning

When using [System.Text.RegularExpressions](#) to process untrusted input, pass a timeout. A malicious user can provide input to `RegularExpression`, causing a [Denial-of-Service attack](#). ASP.NET Core framework APIs that use `RegularExpression` pass a timeout.

Linear comparison without backtracking

If a regular expression pattern has no optional quantifiers or alternation constructs, the regular expression engine executes in linear time. That is, after the regular expression engine matches the first language element in the pattern with text in the input string, it tries to match the next language element in the pattern with the next character or group of characters in the input string. This continues until the match either succeeds or fails. In either case, the regular expression engine advances by one character at a time in the input string.

The following example provides an illustration. The regular expression `e{2}\w\b` looks for two occurrences of the letter "e" followed by any word character followed by a word boundary.

C#

```
using System;
using System.Text.RegularExpressions;

public class Example1
```

```

{
    public static void Run()
    {
        string input = "needing a reed";
        string pattern = @"e{2}\w\b";
        foreach (Match match in Regex.Matches(input, pattern))
            Console.WriteLine("{0} found at position {1}",
                match.Value, match.Index);
    }
}
// The example displays the following output:
//      eed found at position 11

```

Although this regular expression includes the quantifier `{2}`, it is evaluated in a linear manner. The regular expression engine does not backtrack because `{2}` is not an optional quantifier; it specifies an exact number and not a variable number of times that the previous subexpression must match. As a result, the regular expression engine tries to match the regular expression pattern with the input string as shown in the following table.

| Operation | Position in pattern | Position in string | Result |
|------------------|----------------------------|----------------------------|-----------------------|
| 1 | e | "needing a reed" (index 0) | No match. |
| 2 | e | "eeding a reed" (index 1) | Possible match. |
| 3 | e{2} | "eding a reed" (index 2) | Possible match. |
| 4 | \w | "ding a reed" (index 3) | Possible match. |
| 5 | \b | "ing a reed" (index 4) | Possible match fails. |
| 6 | e | "eding a reed" (index 2) | Possible match. |
| 7 | e{2} | "ding a reed" (index 3) | Possible match fails. |
| 8 | e | "ding a reed" (index 3) | Match fails. |
| 9 | e | "ing a reed" (index 4) | No match. |
| 10 | e | "ng a reed" (index 5) | No match. |
| 11 | e | "g a reed" (index 6) | No match. |
| 12 | e | " a reed" (index 7) | No match. |
| 13 | e | "a reed" (index 8) | No match. |
| 14 | e | " reed" (index 9) | No match. |
| 15 | e | "reed" (index 10) | No match |

| Operation | Position in pattern | Position in string | Result |
|------------------|----------------------------|---------------------------|-----------------|
| 16 | e | "eed" (index 11) | Possible match. |
| 17 | e{2} | "ed" (index 12) | Possible match. |
| 18 | \w | "d" (index 13) | Possible match. |
| 19 | \b | "" (index 14) | Match. |

If a regular expression pattern includes no optional quantifiers or alternation constructs, the maximum number of comparisons required to match the regular expression pattern with the input string is roughly equivalent to the number of characters in the input string. In this case, the regular expression engine uses 19 comparisons to identify possible matches in this 13-character string. In other words, the regular expression engine runs in near-linear time if it contains no optional quantifiers or alternation constructs.

Backtracking with optional quantifiers or alternation constructs

When a regular expression includes optional quantifiers or alternation constructs, the evaluation of the input string is no longer linear. Pattern matching with an Nondeterministic Finite Automaton (NFA) engine is driven by the language elements in the regular expression and not by the characters to be matched in the input string. Therefore, the regular expression engine tries to fully match optional or alternative subexpressions. When it advances to the next language element in the subexpression and the match is unsuccessful, the regular expression engine can abandon a portion of its successful match and return to an earlier saved state in the interest of matching the regular expression as a whole with the input string. This process of returning to a previous saved state to find a match is known as backtracking.

For example, consider the regular expression pattern `.*(es)`, which matches the characters "es" and all the characters that precede it. As the following example shows, if the input string is "Essential services are provided by regular expressions.", the pattern matches the whole string up to and including the "es" in "expressions".

C#

```
using System;
using System.Text.RegularExpressions;

public class Example2
{
```

```

public static void Run()
{
    string input = "Essential services are provided by regular
expressions.";
    string pattern = ".*(es)";
    Match m = Regex.Match(input, pattern, RegexOptions.IgnoreCase);
    if (m.Success)
    {
        Console.WriteLine('{0}' found at position {1},
                           m.Value, m.Index);
        Console.WriteLine('es' found at position {0},
                           m.Groups[1].Index);
    }
}
// 'Essential services are provided by regular expressions found at
position 0
// 'es' found at position 47

```

To do this, the regular expression engine uses backtracking as follows:

- It matches the `.*` (which matches zero, one, or more occurrences of any character) with the whole input string.
- It attempts to match "e" in the regular expression pattern. However, the input string has no remaining characters available to match.
- It backtracks to its last successful match, "Essential services are provided by regular expressions", and attempts to match "e" with the period at the end of the sentence. The match fails.
- It continues to backtrack to a previous successful match one character at a time until the tentatively matched substring is "Essential services are provided by regular expr". It then compares the "e" in the pattern to the second "e" in "expressions" and finds a match.
- It compares "s" in the pattern to the "s" that follows the matched "e" character (the first "s" in "expressions"). The match is successful.

When you use backtracking, matching the regular expression pattern with the input string, which is 55 characters long, requires 67 comparison operations. Generally, if a regular expression pattern has a single alternation construct or a single optional quantifier, the number of comparison operations required to match the pattern is more than twice the number of characters in the input string.

Backtracking with nested optional quantifiers

The number of comparison operations required to match a regular expression pattern can increase exponentially if the pattern includes a large number of alternation constructs, if it includes nested alternation constructs, or, most commonly, if it includes nested optional quantifiers. For example, the regular expression pattern `^(a+)+$` is designed to match a complete string that contains one or more "a" characters. The example provides two input strings of identical length, but only the first string matches the pattern. The [System.Diagnostics.Stopwatch](#) class is used to determine how long the match operation takes.

C#

```
using System;
using System.Diagnostics;
using System.Text.RegularExpressions;

public class Example3
{
    public static void Run()
    {
        string pattern = "^(a+)+$";
        string[] inputs = { "aaaaaaaaaaaaaaaaaaaaaaaaaaaaaa",
"aaaaaaaaaaaaaaaaaaaaaaa!" };
        Regex rgx = new Regex(pattern);
        Stopwatch sw;

        foreach (string input in inputs)
        {
            sw = Stopwatch.StartNew();
            Match match = rgx.Match(input);
            sw.Stop();
            if (match.Success)
                Console.WriteLine($"Matched {match.Value} in {sw.Elapsed}");
            else
                Console.WriteLine($"No match found in {sw.Elapsed}");
        }
    }
}

//      Matched aaaaaaaaaaaaaaaaaaaaaaa in 00:00:00.0018281
//      No match found in 00:00:05.1882144
```

As the output from the example shows, the regular expression engine took significantly longer to find that an input string *did not match* the pattern as it did to identify a matching string. This is because an unsuccessful match always represents a worst-case scenario. The regular expression engine must use the regular expression to follow all possible paths through the data before it can conclude that the match is unsuccessful, and the nested parentheses create many additional paths through the data. The regular expression engine concludes that the second string did not match the pattern by doing the following:

- It checks that it was at the beginning of the string, and then matches the first five characters in the string with the pattern `a+`. It then determines that there are no additional groups of "a" characters in the string. Finally, it tests for the end of the string. Because one additional character remains in the string, the match fails. This failed match requires 9 comparisons. The regular expression engine also saves state information from its matches of "a" (which we'll call match 1), "aa" (match 2), "aaa" (match 3), and "aaaa" (match 4).
- It returns to the previously saved match 4. It determines that there is one additional "a" character to assign to an additional captured group. Finally, it tests for the end of the string. Because one additional character remains in the string, the match fails. This failed match requires 4 comparisons. So far, a total of 13 comparisons have been performed.
- It returns to the previously saved match 3. It determines that there are two additional "a" characters to assign to an additional captured group. However, the end-of-string test fails. It then returns to match 3 and tries to match the two additional "a" characters in two additional captured groups. The end-of-string test still fails. These failed matches require 12 comparisons. So far, a total of 25 comparisons have been performed.

Comparison of the input string with the regular expression continues in this way until the regular expression engine has tried all possible combinations of matches, and then concludes that there is no match. Because of the nested quantifiers, this comparison is an $O(2^n)$ or an exponential operation, where n is the number of characters in the input string. This means that in the worst case, an input string of 30 characters requires approximately 1,073,741,824 comparisons, and an input string of 40 characters requires approximately 1,099,511,627,776 comparisons. If you use strings of these or even greater lengths, regular expression methods can take an extremely long time to complete when they process input that does not match the regular expression pattern.

Control backtracking

Backtracking lets you create powerful, flexible regular expressions. However, as the previous section showed, these benefits may be coupled with unacceptably poor performance. To prevent excessive backtracking, you should define a time-out interval when you instantiate a [Regex](#) object or call a static regular expression matching method. This is discussed in the next section. In addition, .NET supports three regular expression language elements that limit or suppress backtracking and that support complex regular expressions with little or no performance penalty: [atomic groups](#), [lookbehind assertions](#),

and lookahead assertions. For more information about each language element, see [Grouping constructs](#).

Non-backtracking regular expression engine

If you don't need to use any constructs that require backtracking (for example, lookarounds, backreferences, or atomic groups), consider using the [RegexOptions.NonBacktracking](#) mode. This mode is designed to execute in time proportional to the length of the input. For more information, see [NonBacktracking mode](#). You can also set a time-out value.

Limit the size of inputs

Some regular expressions have acceptable performance unless the input is exceptionally large. If all reasonable text inputs in your scenario are known to be under a certain length, consider rejecting longer inputs before applying the regular expression to them.

Specify a time-out interval

You can set a time-out value that represents the longest interval the regular expression engine will search for a single match before it abandons the attempt and throws a [RegexMatchTimeoutException](#) exception. You specify the time-out interval by supplying a [TimeSpan](#) value to the [Regex\(String, RegexOptions, TimeSpan\)](#) constructor for instance regular expressions. In addition, each static pattern matching method has an overload with a [TimeSpan](#) parameter that allows you to specify a time-out value.

If you don't set a time-out value explicitly, the default time-out value is determined as follows:

- By using the application-wide time-out value, if one exists. This can be any time-out value that applies to the application domain in which the [Regex](#) object is instantiated or the static method call is made. You can set the application-wide time-out value by calling the [AppDomain.SetData](#) method to assign the string representation of a [TimeSpan](#) value to the "REGEX_DEFAULT_MATCH_TIMEOUT" property.
- By using the value [InfiniteMatchTimeout](#), if no application-wide time-out value has been set.

By default, the time-out interval is set to [Regex.InfiniteMatchTimeout](#) and the regular expression engine does not time out.

Important

When not using [RegexOptions.NonBacktracking](#), we recommend that you always set a time-out interval if your regular expression relies on backtracking or operates on untrusted inputs.

A [RegexMatchTimeoutException](#) exception indicates that the regular expression engine was unable to find a match within the specified time-out interval but does not indicate why the exception was thrown. The reason might be excessive backtracking, but it's also possible that the time-out interval was set too low given the system load at the time the exception was thrown. When you handle the exception, you can choose to abandon further matches with the input string or increase the time-out interval and retry the matching operation.

For example, the following code calls the `Regex(String, RegexOptions, TimeSpan)` constructor to instantiate a `Regex` object with a time-out value of 1 second. The regular expression pattern `(a+)+$`, which matches one or more sequences of one or more "a" characters at the end of a line, is subject to excessive backtracking. If a `RegexMatchTimeoutException` is thrown, the example increases the time-out value up to a maximum interval of 3 seconds. After that, it abandons the attempt to match the pattern.

C#

```

foreach (var inputValue in inputs)
{
    Console.WriteLine("Processing {0}", inputValue);
    bool timedOut = false;
    do
    {
        try
        {
            sw = Stopwatch.StartNew();
            // Display the result.
            if (rgx.IsMatch(inputValue))
            {
                sw.Stop();
                Console.WriteLine(@"Valid: '{0}' ({1:ss\.fffff} seconds)",
                                  inputValue, sw.Elapsed);
            }
            else
            {
                sw.Stop();
                Console.WriteLine(@"'{0}' is not a valid string.
({1:ss\.fffff} seconds)",
                                  inputValue, sw.Elapsed);
            }
        }
        catch (RegexMatchTimeoutException e)
        {
            sw.Stop();
            // Display the elapsed time until the exception.
            Console.WriteLine(@"Timeout with '{0}' after
{1:ss\.fffff}",
                                  inputValue, sw.Elapsed);
            Thread.Sleep(1500);      // Pause for 1.5 seconds.

            // Increase the timeout interval and retry.
            TimeSpan timeout =
e.MatchTimeout.Add(TimeSpan.FromSeconds(1));
            if (timeout.TotalSeconds > MaxTimeoutInSeconds)
            {
                Console.WriteLine("Maximum timeout interval of {0}
seconds exceeded.",
                                  MaxTimeoutInSeconds);
                timedOut = false;
            }
            else
            {
                Console.WriteLine("Changing the timeout interval to
{0}",
                                  timeout);
                rgx = new Regex(pattern, RegexOptions.IgnoreCase,
timeout);
                timedOut = true;
            }
        }
    } while (timedOut);
}

```

```

        Console.WriteLine();
    }
}

// The example displays output like the following :
// Processing aa
// Valid: 'aa' (00.0000779 seconds)
//
// Processing aaaa>
// 'aaaa>' is not a valid string. (00.00005 seconds)
//
// Processingaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
// Valid: 'aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa' (00.0000043
seconds)
//
// Processingaaaaaaaaaaaaaaaaaaaa>
// Timeout with 'aaaaaaaaaaaaaaaaaaaa' after 01.00469
// Changing the timeout interval to 00:00:02
// Timeout with 'aaaaaaaaaaaaaaaaaaaa' after 02.01202
// Changing the timeout interval to 00:00:03
// Timeout with 'aaaaaaaaaaaaaaaaaaaa' after 03.01043
// Maximum timeout interval of 3 seconds exceeded.
//
// Processingaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa>
// Timeout with 'aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa' after
03.01018
// Maximum timeout interval of 3 seconds exceeded.

```

Atomic groups

The `(?>subexpression)` language element is an atomic grouping. It prevents backtracking into the subexpression. Once this language element has successfully matched, it will not give up any part of its match to subsequent backtracking. For example, in the pattern `(?>\w*\d*)1`, if the `1` cannot be matched, the `\d*` will not give up any of its match even if that means it would allow the `1` to successfully match. Atomic groups can help prevent the performance problems associated with failed matches.

The following example illustrates how suppressing backtracking improves performance when using nested quantifiers. It measures the time required for the regular expression engine to determine that an input string does not match two regular expressions. The first regular expression uses backtracking to attempt to match a string that contains one or more occurrences of one or more hexadecimal digits, followed by a colon, followed by one or more hexadecimal digits, followed by two colons. The second regular expression is identical to the first, except that it disables backtracking. As the output from the example shows, the performance improvement from disabling backtracking is significant.

C#

```
using System;
using System.Diagnostics;
using System.Text.RegularExpressions;

public class Example4
{
    public static void Run()
    {
        string input =
"b51:4:1DB:9EE1:5:27d60:f44:D4:cd:E:5:0A5:4a:D24:41Ad:";
        bool matched;
        Stopwatch sw;

        Console.WriteLine("With backtracking:");
        string backPattern = "^(([0-9a-fA-F]{1,4}:)*([0-9a-fA-F]{1,4}))*(:::$";
        sw = Stopwatch.StartNew();
        matched = Regex.IsMatch(input, backPattern);
        sw.Stop();
        Console.WriteLine("Match: {0} in {1}", Regex.IsMatch(input,
backPattern), sw.Elapsed);
        Console.WriteLine();

        Console.WriteLine("Without backtracking:");
        string noBackPattern = "^(?>[0-9a-fA-F]{1,4}:)*(?>[0-9a-fA-F]
{1,4}))*(:::$";
        sw = Stopwatch.StartNew();
        matched = Regex.IsMatch(input, noBackPattern);
        sw.Stop();
        Console.WriteLine("Match: {0} in {1}", Regex.IsMatch(input,
noBackPattern), sw.Elapsed);
    }
}

// The example displays output like the following:
//      With backtracking:
//          Match: False in 00:00:27.4282019
//
//      Without backtracking:
//          Match: False in 00:00:00.0001391
```

Lookbehind assertions

.NET includes two language elements, `(?<=subexpression)` and `(?<!subexpression)`, that match the previous character or characters in the input string. Both language elements are zero-width assertions; that is, they determine whether the character or characters that immediately precede the current character can be matched by *subexpression*, without advancing or backtracking.

`(?<=subexpression)` is a positive lookbehind assertion; that is, the character or characters before the current position must match *subexpression*. `(?<!subexpression)` is a negative lookbehind assertion; that is, the character or characters before the current position must not match *subexpression*. Both positive and negative lookbehind assertions are most useful when *subexpression* is a subset of the previous subexpression.

The following example uses two equivalent regular expression patterns that validate the user name in an email address. The first pattern is subject to poor performance because of excessive backtracking. The second pattern modifies the first regular expression by replacing a nested quantifier with a positive lookbehind assertion. The output from the example displays the execution time of the [Regex.IsMatch](#) method.

C#

```
using System;
using System.Diagnostics;
using System.Text.RegularExpressions;

public class Example5
{
    public static void Run()
    {
        Stopwatch sw;
        string input = "test@contoso.com";
        bool result;

        string pattern = @"^[\w]+@[.\w]*[\w]+@[.\w]+";
        sw = Stopwatch.StartNew();
        result = Regex.IsMatch(input, pattern, RegexOptions.IgnoreCase);
        sw.Stop();
        Console.WriteLine("Match: {0} in {1}", result, sw.Elapsed);

        string behindPattern = @"^[\w]+@[.\w]*(?<=[\w]+)@";
        sw = Stopwatch.StartNew();
        result = Regex.IsMatch(input, behindPattern,
        RegexOptions.IgnoreCase);
        sw.Stop();
        Console.WriteLine("Match with Lookbehind: {0} in {1}", result,
        sw.Elapsed);
    }
}

// The example displays output similar to the following:
//      Match: True in 00:00:00.0017549
//      Match with Lookbehind: True in 00:00:00.0000659
```

The first regular expression pattern, `^[\w]+@[.\w]*[\w]+@[.\w]+`, is defined as shown in the following table.

| Pattern | Description |
|--------------------|---|
| ^ | Start the match at the beginning of the string. |
| [0-9A-Z] | Match an alphanumeric character. This comparison is case-insensitive, because the Regex.IsMatch method is called with the RegexOptions.IgnoreCase option. |
| [-.\w]* | Match zero, one, or more occurrences of a hyphen, period, or word character. |
| [0-9A-Z] | Match an alphanumeric character. |
| ([-.\w]*[0-9A-Z])* | Match zero or more occurrences of the combination of zero or more hyphens, periods, or word characters, followed by an alphanumeric character. This is the first capturing group. |
| @ | Match an at sign ("@"). |

The second regular expression pattern, `^[0-9A-Z][-.\w]*(?=[0-9A-Z])@`, uses a positive lookbehind assertion. It is defined as shown in the following table.

| Pattern | Description |
|--------------|---|
| ^ | Start the match at the beginning of the string. |
| [0-9A-Z] | Match an alphanumeric character. This comparison is case-insensitive, because the Regex.IsMatch method is called with the RegexOptions.IgnoreCase option. |
| [-.\w]* | Match zero or more occurrences of a hyphen, period, or word character. |
| (?=[0-9A-Z]) | Look back at the last matched character and continue the match if it is alphanumeric. Note that alphanumeric characters are a subset of the set that consists of periods, hyphens, and all word characters. |
| @ | Match an at sign ("@"). |

Lookahead assertions

.NET includes two language elements, `(?=subexpression)` and `(?!subexpression)`, that match the next character or characters in the input string. Both language elements are zero-width assertions; that is, they determine whether the character or characters that immediately follow the current character can be matched by *subexpression*, without advancing or backtracking.

`(?=subexpression)` is a positive lookahead assertion; that is, the character or characters after the current position must match *subexpression*. `(?!subexpression)` is a negative lookahead assertion; that is, the character or characters after the current position must

not match *subexpression*. Both positive and negative lookahead assertions are most useful when *subexpression* is a subset of the next subexpression.

The following example uses two equivalent regular expression patterns that validate a fully qualified type name. The first pattern is subject to poor performance because of excessive backtracking. The second modifies the first regular expression by replacing a nested quantifier with a positive lookahead assertion. The output from the example displays the execution time of the `Regex.IsMatch` method.

C#

```
using System;
using System.Diagnostics;
using System.Text.RegularExpressions;

public class Example6
{
    public static void Run()
    {
        string input = "aaaaaaaaaaaaaaaaaaaaaa.";
        bool result;
        Stopwatch sw;

        string pattern = @"^(([A-Z]\w*)+\.)*[A-Z]\w*$/";
        sw = Stopwatch.StartNew();
        result = Regex.IsMatch(input, pattern, RegexOptions.IgnoreCase);
        sw.Stop();
        Console.WriteLine("{0} in {1}", result, sw.Elapsed);

        string aheadPattern = @"^((?=([A-Z])\w+)\.)*[A-Z]\w*$/";
        sw = Stopwatch.StartNew();
        result = Regex.IsMatch(input, aheadPattern,
        RegexOptions.IgnoreCase);
        sw.Stop();
        Console.WriteLine("{0} in {1}", result, sw.Elapsed);
    }
}
// The example displays the following output:
//      False in 00:00:03.8003793
//      False in 00:00:00.0000866
```

The first regular expression pattern, `^(([A-Z]\w*)+\.)*[A-Z]\w*$/`, is defined as shown in the following table.

| Pattern | Description |
|-------------------------------|--|
| <code>^</code> | Start the match at the beginning of the string. |
| <code>(([A-Z]\w*)+\.)*</code> | Match an alphabetical character (A-Z) followed by zero or more word characters one or more times, followed by a period. This comparison is case-insensitive, |

| Pattern | Description |
|---------------------------------|---|
| <code>(([A-Z]\w*)+\.\.)*</code> | Match the previous pattern zero or more times. |
| <code>[A-Z]\w*</code> | Match an alphabetical character followed by zero or more word characters. |
| <code>\$</code> | End the match at the end of the input string. |

The second regular expression pattern, `^((?=([A-Z])\w+\.\.)*[A-Z]\w*)$`, uses a positive lookahead assertion. It is defined as shown in the following table.

| Pattern | Description |
|-----------------------------------|---|
| <code>^</code> | Start the match at the beginning of the string. |
| <code>(?=([A-Z])</code> | Look ahead to the first character and continue the match if it is alphabetical (A-Z). This comparison is case-insensitive, because the Regex.IsMatch method is called with the RegexOptions.IgnoreCase option. |
| <code>\w+\.\.</code> | Match one or more word characters followed by a period. |
| <code>((?=([A-Z])\w+\.\.)*</code> | Match the pattern of one or more word characters followed by a period zero or more times. The initial word character must be alphabetical. |
| <code>[A-Z]\w*</code> | Match an alphabetical character followed by zero or more word characters. |
| <code>\$</code> | End the match at the end of the input string. |

General performance considerations

The following suggestions are not specifically to prevent excessive backtracking, but may help increase the performance of your regular expression:

1. Precompile heavily used patterns. The best way to do this is to use the [regular expression source generator](#) to precompile it. If the source generator is not available for your app, for example you are not targeting .NET 7 or later, or you do not know the pattern at compile time, use the [RegexOptions.Compiled](#) option.
2. Cache heavily used Regex objects. This implicitly occurs when you are using the source generator. Otherwise, create a Regex object and store it for reuse, rather than using the static Regex methods or creating and throwing away a Regex object.
3. Start matching from an offset. If you know that matches will always start beyond a certain offset into the pattern, pass the offset in using an overload like

`Regex.Match(String, Int32)`. This will reduce the amount of the text the engine needs to consider.

4. Gather only the information you need. If you only need to know whether a match occurs but not where the match occurs, prefer `Regex.IsMatch`. If you only need to know how many times something matches, prefer using `Regex.Count`. If you only need to know the bounds of a match but not anything about a match's captures, prefer using `Regex.EnumerateMatches`. The less information the engine needs to provide, the better.
5. Avoid unnecessary captures. Parentheses in your pattern form a capturing group by default. If you don't need captures, either specify `RegexOptions.ExplicitCapture` or use [non-capturing groups](#) instead. This saves the engine keeping track of those captures.

See also

- [.NET Regular Expressions](#)
- [Regular Expression Language - Quick Reference](#)
- [Quantifiers](#)
- [Alternation Constructs](#)
- [Grouping Constructs](#)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

Compilation and Reuse in Regular Expressions

Article • 09/15/2021

You can optimize the performance of applications that make extensive use of regular expressions by understanding how the regular expression engine compiles expressions and by understanding how regular expressions are cached. This topic discusses both compilation and caching.

Compiled Regular Expressions

By default, the regular expression engine compiles a regular expression to a sequence of internal instructions (these are high-level codes that are different from Microsoft intermediate language, or MSIL). When the engine executes a regular expression, it interprets the internal codes.

If a [Regex](#) object is constructed with the [RegexOptions.Compiled](#) option, it compiles the regular expression to explicit MSIL code instead of high-level regular expression internal instructions. This allows .NET's just-in-time (JIT) compiler to convert the expression to native machine code for higher performance. The cost of constructing the [Regex](#) object may be higher, but the cost of performing matches with it is likely to be much smaller.

An alternative is to use precompiled regular expressions. You can compile all of your expressions into a reusable DLL by using the [CompileToAssembly](#) method. This avoids the need to compile at run time while still benefiting from the speed of compiled regular expressions.

The Regular Expressions Cache

To improve performance, the regular expression engine maintains an application-wide cache of compiled regular expressions. The cache stores regular expression patterns that are used only in static method calls. (Regular expression patterns supplied to instance methods are not cached.) This avoids the need to reparse an expression into high-level byte code each time it is used.

The maximum number of cached regular expressions is determined by the value of the [static](#) (Shared in Visual Basic) [Regex.CacheSize](#) property. By default, the regular expression engine caches up to 15 compiled regular expressions. If the number of

compiled regular expressions exceeds the cache size, the least recently used regular expression is discarded and the new regular expression is cached.

Your application can reuse regular expressions in one of the following two ways:

- By using a static method of the [Regex](#) object to define the regular expression. If you're using a regular expression pattern that has already been defined by another static method call, the regular expression engine will try to retrieve it from the cache. If it's not available in the cache, the engine will compile the regular expression and add it to the cache.
- By reusing an existing [Regex](#) object as long as its regular expression pattern is needed.

Because of the overhead of object instantiation and regular expression compilation, creating and rapidly destroying numerous [Regex](#) objects is a very expensive process. For applications that use a large number of different regular expressions, you can optimize performance by using calls to static [Regex](#) methods and possibly by increasing the size of the regular expression cache.

See also

- [.NET Regular Expressions](#)



Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



.NET feedback

The .NET documentation is open source. Provide feedback here.



[Open a documentation issue](#)



[Provide product feedback](#)

Thread Safety in Regular Expressions

Article • 09/15/2021

The [Regex](#) class itself is thread safe and immutable (read-only). That is, **Regex** objects can be created on any thread and shared between threads; matching methods can be called from any thread and never alter any global state.

However, result objects (**Match** and **MatchCollection**) returned by **Regex** should be used on a single thread. Although many of these objects are logically immutable, their implementations could delay computation of some results to improve performance, and as a result, callers must serialize access to them.

If there is a need to share **Regex** result objects on multiple threads, these objects can be converted to thread-safe instances by calling their synchronized methods. With the exception of enumerators, all regular expression classes are thread safe or can be converted into thread-safe objects by a synchronized method.

Enumerators are the only exception. An application must serialize calls to collection enumerators. The rule is that if a collection can be enumerated on more than one thread simultaneously, you should synchronize enumerator methods on the root object of the collection traversed by the enumerator.

See also

- [.NET Regular Expressions](#)

 Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback [here](#).

 [Open a documentation issue](#)

 [Provide product feedback](#)

Regular expression example: Scanning for HREFs

Article • 09/15/2021

The following example searches an input string and displays all the href="..." values and their locations in the string.

⚠ Warning

When using `System.Text.RegularExpressions` to process untrusted input, pass a timeout. A malicious user can provide input to `RegularExpressions`, causing a **Denial-of-Service attack** [↗](#). ASP.NET Core framework APIs that use `RegularExpressions` pass a timeout.

The Regex object

Because the `DumpHRefs` method can be called multiple times from user code, it uses the `static` (`Shared` in Visual Basic) `Regex.Match(String, String, RegexOptions)` method. This enables the regular expression engine to cache the regular expression and avoids the overhead of instantiating a new `Regex` object each time the method is called. A `Match` object is then used to iterate through all matches in the string.

C#

```
private static void DumpHRefs(string inputString)
{
    string hrefPattern = @"href\s*=\s*(?:['"](?<1>[^'"]*)['"]|(?<1>
[^>\s]+))";
}

try
{
    Match regexMatch = Regex.Match(inputString, hrefPattern,
                                   RegexOptions.IgnoreCase |
    RegexOptions.Compiled,
                                   TimeSpan.FromSeconds(1));
    while (regexMatch.Success)
    {
        Console.WriteLine($"Found href {regexMatch.Groups[1]} at
{regexMatch.Groups[1].Index}");
        regexMatch = regexMatch.NextMatch();
    }
}
catch (RegexMatchTimeoutException)
```

```

    {
        Console.WriteLine("The matching operation timed out.");
    }
}

```

The following example then illustrates a call to the `DumpHRefs` method.

C#

```

public static void Main()
{
    string inputString = "My favorite web sites include:</P>" +
        "<A HREF=\"https://learn.microsoft.com/en-
us/dotnet/\">>" +
        ".NET Documentation</A></P>" +
        "<A HREF=\"http://www.microsoft.com\>" +
        "Microsoft Corporation Home Page</A></P>" +
        "<A
HREF=\"https://devblogs.microsoft.com/dotnet/\>" +
        ".NET Blog</A></P>";
    DumpHRefs(inputString);
}

// The example displays the following output:
//      Found href https://learn.microsoft.com/dotnet/ at 43
//      Found href http://www.microsoft.com at 114
//      Found href https://devblogs.microsoft.com/dotnet/ at 188

```

The regular expression pattern `href\s*=\s*(?:['"](?<1>[^'"]*)['"]|(?<1>[^>\s]+))` is interpreted as shown in the following table.

| Pattern | Description |
|--|---|
| <code>href</code> | Match the literal string "href". The match is case-insensitive. |
| <code>\s*</code> | Match zero or more white-space characters. |
| <code>=</code> | Match the equals sign. |
| <code>\s*</code> | Match zero or more white-space characters. |
| <code>(?:</code> | Start a non-capturing group. |
| <code>['"](?<1>[^'"]*)['"] (?<1>[^>\s]+)</code> | Match a quotation mark or apostrophe, followed by a capturing group that matches any character other than a quotation mark or apostrophe, followed by a quotation mark or apostrophe. The group named <code>1</code> is included in this pattern. |
| <code> </code> | Boolean OR that matches either the previous expression or the next expression. |
| <code>(?<1>[^>\s]+)</code> | A capturing group that uses a negated set to match any character other than a greater-than sign or a whitespace character. The group named <code>1</code> is included in |

| Pattern | Description |
|---------|------------------------------|
| | this pattern. |
|) | End the non-capturing group. |

Match result class

The results of a search are stored in the [Match](#) class, which provides access to all the substrings extracted by the search. It also remembers the string being searched and the regular expression being used, so it can call the [Match.NextMatch](#) method to perform another search starting where the last one ended.

Explicitly named captures

In traditional regular expressions, capturing parentheses are automatically numbered sequentially. This leads to two problems. First, if a regular expression is modified by inserting or removing a set of parentheses, all code that refers to the numbered captures must be rewritten to reflect the new numbering. Second, because different sets of parentheses often are used to provide two alternative expressions for an acceptable match, it might be difficult to determine which of the two expressions actually returned a result.

To address these problems, the [Regex](#) class supports the syntax `(?<name>...)` for capturing a match into a specified slot (the slot can be named using a string or an integer; integers can be recalled more quickly). Thus, alternative matches for the same string all can be directed to the same place. In case of a conflict, the last match dropped into a slot is the successful match. (However, a complete list of multiple matches for a single slot is available. See the [Group.Captures](#) collection for details.)

See also

- [.NET Regular Expressions](#)

 Collaborate with us on
GitHub

The source for this content can
be found on GitHub, where you
can also create and review

.NET

.NET feedback

The .NET documentation is open
source. Provide feedback here.

issues and pull requests. For more information, see [our contributor guide](#).

 Open a documentation issue

 Provide product feedback

Regular Expression Example: Changing Date Formats

Article • 09/15/2021

The following code example uses the `Regex.Replace` method to replace dates that have the form *mm/dd/yy* with dates that have the form *dd-mm-yy*.

⚠ Warning

When using `System.Text.RegularExpressions` to process untrusted input, pass a timeout. A malicious user can provide input to `RegularExpressions`, causing a [Denial-of-Service attack](#). ASP.NET Core framework APIs that use `RegularExpressions` pass a timeout.

Example

C#

```
static string MDYToDMY(string input)
{
    try {
        return Regex.Replace(input,
            @"\b(?:month)\d{1,2})/(?:day)\d{1,2})/(?:year)\d{2,4})\b",
            "${day}-${month}-${year}", RegexOptions.None,
            TimeSpan.FromMilliseconds(150));
    }
    catch (RegexMatchTimeoutException) {
        return input;
    }
}
```

The following code shows how the `MDYToDMY` method can be called in an application.

C#

```
using System;
using System.Globalization;
using System.Text.RegularExpressions;

public class Class1
{
    public static void Main()
    {
```

```

        string dateString = DateTime.Today.ToString("d",
                                                DateTimeFormatInfo.InvariantInfo);
        string resultString = MDYToDMY(dateString);
        Console.WriteLine("Converted {0} to {1}.", dateString, resultString);
    }

    static string MDYToDMY(string input)
    {
        try {
            return Regex.Replace(input,
                @"\b(?<month>\d{1,2})/(\?<day>\d{1,2})/(\?<year>\d{2,4})\b",
                "${day}-${month}-${year}", RegexOptions.None,
                TimeSpan.FromMilliseconds(150));
        }
        catch (RegexMatchTimeoutException) {
            return input;
        }
    }
}

// The example displays the following output to the console if run on
// 8/21/2007:
//      Converted 08/21/2007 to 21-08-2007.

```

Comments

The regular expression pattern `\b(?<month>\d{1,2})/(\?<day>\d{1,2})/(\?<year>\d{2,4})\b` is interpreted as shown in the following table.

| Pattern | Description |
|--------------------------------------|--|
| <code>\b</code> | Begin the match at a word boundary. |
| <code>(?<month>\d{1,2})</code> | Match one or two decimal digits. This is the <code>month</code> captured group. |
| <code>/</code> | Match the slash mark. |
| <code>(?<day>\d{1,2})</code> | Match one or two decimal digits. This is the <code>day</code> captured group. |
| <code>/</code> | Match the slash mark. |
| <code>(?<year>\d{2,4})</code> | Match from two to four decimal digits. This is the <code>year</code> captured group. |
| <code>\b</code> | End the match at a word boundary. |

The pattern `${day}-${month}-${year}` defines the replacement string as shown in the following table.

| Pattern | Description |
|------------------------|--|
| <code>\$(day)</code> | Add the string captured by the <code>day</code> capturing group. |
| <code>-</code> | Add a hyphen. |
| <code>\$(month)</code> | Add the string captured by the <code>month</code> capturing group. |
| <code>-</code> | Add a hyphen. |
| <code>\$(year)</code> | Add the string captured by the <code>year</code> capturing group. |

See also

- [.NET Regular Expressions](#)

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

How to: Extract a Protocol and Port Number from a URL

Article • 09/15/2021

The following example extracts a protocol and port number from a URL.

⚠ Warning

When using `System.Text.RegularExpressions` to process untrusted input, pass a timeout. A malicious user can provide input to `RegularExpressions`, causing a [Denial-of-Service attack](#). ASP.NET Core framework APIs that use `RegularExpressions` pass a timeout.

Example

The example uses the `Match.Result` method to return the protocol followed by a colon followed by the port number.

C#

```
using System;
using System.Text.RegularExpressions;

public class Example
{
    public static void Main()
    {
        string url = "http://www.contoso.com:8080/letters/readme.html";

        Regex r = new Regex(@"^(?<proto>\w+://[^/]+?(?<port>:\d+)?/");
        RegexOptions.None,
        TimeSpan.FromMilliseconds(150));
        Match m = r.Match(url);
        if (m.Success)
            Console.WriteLine(m.Result("${proto}${port}"));
    }
}
// The example displays the following output:
//      http:8080
```

The regular expression pattern `^(?<proto>\w+://[^/]+?(?<port>:\d+)?/` can be interpreted as shown in the following table.

| Pattern | Description |
|--|---|
| <code>^</code> | Begin the match at the start of the string. |
| <code>(?<proto>\w+)</code> | Match one or more word characters. Name this group <code>proto</code> . |
| <code>::/</code> | Match a colon followed by two slash marks. |
| <code>[^/]+?</code> | Match one or more occurrences (but as few as possible) of any character other than a slash mark. |
| <code>(?<port>:\d+)?< code=""></port>:\d+)?<></code> | Match zero or one occurrence of a colon followed by one or more digit characters. Name this group <code>port</code> . |
| <code>/</code> | Match a slash mark. |

The [Match.Result](#) method expands the `${proto}${port}` replacement sequence, which concatenates the value of the two named groups captured in the regular expression pattern. It is a convenient alternative to explicitly concatenating the strings retrieved from the collection object returned by the [Match.Groups](#) property.

The example uses the [Match.Result](#) method with two substitutions, `${proto}` and `${port}`, to include the captured groups in the output string. You can retrieve the captured groups from the match's [GroupCollection](#) object instead, as the following code shows.

C#

```
Console.WriteLine(m.Groups["proto"].Value + m.Groups["port"].Value);
```

See also

- [.NET Regular Expressions](#)

 Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

more information, see [our contributor guide](#).

How to: Strip Invalid Characters from a String

Article • 09/15/2021

The following example uses the static `Regex.Replace` method to strip invalid characters from a string.

⚠ Warning

When using `System.Text.RegularExpressions` to process untrusted input, pass a timeout. A malicious user can provide input to `RegularExpressions`, causing a **Denial-of-Service attack**. ASP.NET Core framework APIs that use `RegularExpressions` pass a timeout.

Example

You can use the `CleanInput` method defined in this example to strip potentially harmful characters that have been entered into a text field that accepts user input. In this case, `CleanInput` strips out all nonalphanumeric characters except periods (.), at symbols (@), and hyphens (-), and returns the remaining string. However, you can modify the regular expression pattern so that it strips out any characters that should not be included in an input string.

C#

```
using System;
using System.Text.RegularExpressions;

public class Example
{
    static string CleanInput(string strIn)
    {
        // Replace invalid characters with empty strings.
        try {
            return Regex.Replace(strIn, @"[^\\w\\.@-]", "", RegexOptions.None,
TimeSpan.FromSeconds(1.5));
        }
        // If we timeout when replacing invalid characters,
        // we should return Empty.
        catch (RegexMatchTimeoutException) {
            return String.Empty;
        }
    }
}
```

```
    }  
}
```

The regular expression pattern `[^\w\.\@-]` matches any character that is not a word character, a period, an @ symbol, or a hyphen. A word character is any letter, decimal digit, or punctuation connector such as an underscore. Any character that matches this pattern is replaced by `String.Empty`, which is the string defined by the replacement pattern. To allow additional characters in user input, add those characters to the character class in the regular expression pattern. For example, the regular expression pattern `[^\w\.\@-\%\]` also allows a percentage symbol and a backslash in an input string.

See also

- [.NET Regular Expressions](#)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

How to verify that strings are in valid email format

Article • 10/04/2022

The example in this article uses a regular expression to verify that a string is in valid email format.

This regular expression is comparatively simple to what can actually be used as an email. Using a regular expression to validate an email is useful to ensure that the structure of an email is correct. However, it isn't a substitution for verifying the email actually exists.

- ✓ DO use a small regular expression to check for the valid structure of an email.
- ✓ DO send a test email to the address provided by a user of your app.
- ✗ DON'T use a regular expression as the only way you validate an email.

If you try to create the *perfect* regular expression to validate that the structure of an email is correct, the expression becomes so complex that it's incredibly difficult to debug or improve. Regular expressions can't validate an email exists, even if it's structured correctly. The best way to validate an email is to send a test email to the address.

⚠ Warning

When using `System.Text.RegularExpressions` to process untrusted input, pass a timeout. A malicious user can provide input to `RegularExpressions`, causing a [Denial-of-Service attack](#). ASP.NET Core framework APIs that use `RegularExpressions` pass a timeout.

Example

The example defines an `IsValidEmail` method, which returns `true` if the string contains a valid email address and `false` if it doesn't but takes no other action.

To verify that the email address is valid, the `IsValidEmail` method calls the `Regex.Replace(String, String, MatchEvaluator)` method with the `(@)(.+)$` regular expression pattern to separate the domain name from the email address. The third parameter is a `MatchEvaluator` delegate that represents the method that processes and replaces the matched text. The regular expression pattern is interpreted as follows:

| Pattern | Description |
|---------|--|
| (@) | Match the @ character. This part is the first capturing group. |
| (.+) | Match one or more occurrences of any character. This part is the second capturing group. |
| \$ | End the match at the end of the string. |

The domain name, along with the @ character, is passed to the `DomainMapper` method. The method uses the `IdnMapping` class to translate Unicode characters that are outside the US-ASCII character range to Punycode. The method also sets the `invalid` flag to `True` if the `IdnMapping.GetAscii` method detects any invalid characters in the domain name. The method returns the Punycode domain name preceded by the @ symbol to the `IsValidEmail` method.

💡 Tip

It's recommended that you use the simple `(@)(.+)$` regular expression pattern to normalize the domain and then return a value indicating that it passed or failed. However, the example in this article describes how to use a regular expression further to validate the email. Regardless of how you validate an email, you should always send a test email to the address to ensure it exists.

The `IsValidEmail` method then calls the `Regex.IsMatch(String, String)` method to verify that the address conforms to a regular expression pattern.

The `IsValidEmail` method merely determines whether the email format is valid for an email address; it doesn't validate that the email exists. Also, the `IsValidEmail` method doesn't verify that the top-level domain name is a valid domain name listed in the [IANA Root Zone Database](#), which would require a look-up operation.

C#

```
using System;
using System.Globalization;
using System.Text.RegularExpressions;

namespace RegexExamples
{
    class RegexUtilities
    {
        public static bool IsValidEmail(string email)
        {
            if (string.IsNullOrWhiteSpace(email))
                return false;
            else
                return Regex.IsMatch(email, "(@)(.+)$");
        }
    }
}
```

```

        return false;

    try
    {
        // Normalize the domain
        email = Regex.Replace(email, @"(@)(.+)$", DomainMapper,
            RegexOptions.None,
        TimeSpan.FromMilliseconds(200));

        // Examines the domain part of the email and normalizes it.
        string DomainMapper(Match match)
        {
            // Use IdnMapping class to convert Unicode domain names.
            var idn = new IdnMapping();

            // Pull out and process domain name (throws
            ArgumentException on invalid)
            string domainName = idn.GetAscii(match.Groups[2].Value);

            return match.Groups[1].Value + domainName;
        }
    }
    catch (RegexMatchTimeoutException e)
    {
        return false;
    }
    catch (ArgumentException e)
    {
        return false;
    }

    try
    {
        return Regex.IsMatch(email,
            @"^[@\s]+@[^\s]+\.[^\s]+$",
            RegexOptions.IgnoreCase,
        TimeSpan.FromMilliseconds(250));
    }
    catch (RegexMatchTimeoutException)
    {
        return false;
    }
}
}

```

In this example, the regular expression pattern `^[@\s]+@[^\s]+\.[^\s]+$` is interpreted as shown in the following table. The regular expression is compiled using the `RegexOptions.IgnoreCase` flag.

| Pattern | Description |
|----------------------|--|
| <code>^</code> | Begin the match at the start of the string. |
| <code>[^@\s]+</code> | Match one or more occurrences of any character other than the @ character or whitespace. |
| <code>@</code> | Match the @ character. |
| <code>[^@\s]+</code> | Match one or more occurrences of any character other than the @ character or whitespace. |
| <code>\.</code> | Match a single period character. |
| <code>[^@\s]+</code> | Match one or more occurrences of any character other than the @ character or whitespace. |
| <code>\$</code> | End the match at the end of the string. |

ⓘ Important

This regular expression isn't intended to cover every aspect of a valid email address. It's provided as an example for you to extend as needed.

See also

- [.NET Regular Expressions](#)
- [How far should one take e-mail address validation? ↗](#)

ⓘ Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

ⓘ .NET feedback

The .NET documentation is open source. Provide feedback here.

[ⓘ Open a documentation issue](#)

[ⓘ Provide product feedback](#)

Serialization in .NET

Article • 10/25/2023

Serialization is the process of converting the state of an object into a form that can be persisted or transported. The complement of serialization is deserialization, which converts a stream into an object. Together, these processes allow data to be stored and transferred.

.NET features the following serialization technologies:

- [JSON serialization](#) maps .NET objects to and from JavaScript Object Notation (JSON). JSON is an open standard that's commonly used to share data across the web. The JSON serializer serializes public properties by default, and can be configured to serialize private and internal members as well.
- [XML and SOAP serialization](#) serializes only `public` properties and fields and does not preserve type fidelity. This is useful when you want to provide or consume data without restricting the application that uses the data. Because XML is an open standard, it is an attractive choice for sharing data across the Web. SOAP is likewise an open standard, which makes it an attractive choice.
- [Binary serialization](#) preserves *type fidelity*, which means that the complete state of the object is recorded and when you deserialize, an exact copy is created. This type of serialization is useful for preserving the state of an object between different invocations of an application. For example, you can share an object between different applications by serializing it to the Clipboard. You can serialize an object to a stream, to a disk, to memory, over the network, and so forth. Remoting uses serialization to pass objects "by value" from one computer or application domain to another.

⚠ Warning

Binary serialization can be dangerous. For more information, see [BinaryFormatter security guide](#).

Reference

[System.Text.Json](#)

Contains classes that can be used to serialize objects into JSON format documents or streams.

[System.Runtime.Serialization](#)

Contains classes that can be used for serializing and deserializing objects.

[System.Xml.Serialization](#)

Contains classes that can be used to serialize objects into XML format documents or streams.

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

JSON serialization and deserialization (marshalling and unmarshalling) in .NET - overview

Article • 10/25/2023

The [System.Text.Json](#) namespace provides functionality for serializing to and deserializing from JavaScript Object Notation (JSON). *Serialization* is the process of converting the state of an object, that is, the values of its properties, into a form that can be stored or transmitted. The serialized form doesn't include any information about an object's associated methods. *Deserialization* reconstructs an object from the serialized form.

The `System.Text.Json` library design emphasizes high performance and low memory allocation over an extensive feature set. Built-in UTF-8 support optimizes the process of reading and writing JSON text encoded as UTF-8, which is the most prevalent encoding for data on the web and files on disk.

The library also provides classes for working with an in-memory [document object model \(DOM\)](#). This feature enables random access to the elements in a JSON file or string.

For Visual Basic, there are some limitations on what parts of the library you can use. For more information, see [Visual Basic support](#).

How to get the library

The library is built-in as part of the shared framework for .NET Core 3.0 and later versions. The [source generation feature](#) is built-in as part of the shared framework for .NET 6 and later versions.

For framework versions earlier than .NET Core 3.0, install the [System.Text.Json](#) NuGet package. The package supports:

- .NET Standard 2.0 and later
- .NET Framework 4.6.2 and later
- .NET Core 2.1 and later
- .NET 5 and later

Namespaces and APIs

- The [System.Text.Json](#) namespace contains all the entry points and the main types.
- The [System.Text.Json.Serialization](#) namespace contains attributes and APIs for advanced scenarios and customization specific to serialization and deserialization.

The code examples shown in this article require `using` directives for one or both of these namespaces.

Important

`System.Text.Json` doesn't support the following serialization APIs that you might have used previously:

- Attributes from the [System.Runtime.Serialization](#) namespace.
 - The [System.SerializableAttribute](#) attribute and the [ISerializable](#) interface.
- These types are used only for [Binary](#) and [XML](#) serialization.

HttpClient and HttpContent extension methods

Serializing and deserializing JSON payloads from the network are common operations. Extension methods on [HttpClient](#) and [HttpContent](#) let you do these operations in a single line of code. These extension methods use [web defaults for JsonSerializerOptions](#).

The following example illustrates use of [HttpClientJsonExtensions.GetFromJsonAsync](#) and [HttpClientJsonExtensions.PostAsJsonAsync](#):

C#

```
using System.Net.Http.Json;

namespace HttpClientExtensionMethods
{
    public class User
    {
        public int Id { get; set; }
        public string? Name { get; set; }
        public string? Username { get; set; }
        public string? Email { get; set; }
    }

    public class Program
    {
        public static async Task Main()
        {
            using HttpClient client = new()
            {
                BaseAddress = new
            }
        }
    }
}
```

```

Uri("https://jsonplaceholder.typicode.com")
};

// Get the user information.
User? user = await client.GetFromJsonAsync<User>("users/1");
Console.WriteLine($"Id: {user?.Id}");
Console.WriteLine($"Name: {user?.Name}");
Console.WriteLine($"Username: {user?.Username}");
Console.WriteLine($"Email: {user?.Email}");

// Post a new user.
HttpResponseMessage response = await
client.PostAsJsonAsync("users", user);
Console.WriteLine(
 $"{(response.IsSuccessStatusCode ? "Success" : "Error")} - 
{response.StatusCode}");
}
}

// Produces output like the following example but with different names:
// 
//Id: 1
//Name: Tyler King
//Username: Tyler
//Email: Tyler @contoso.com
//Success - Created

```

There are also extension methods for `System.Text.Json` on `HttpContent`.

Reflection vs. source generation

By default, `System.Text.Json` gathers the metadata it needs to access properties of objects for serialization and deserialization *at run time* using [reflection](#). As an alternative, `System.Text.Json` can use the C# [source generation](#) feature to improve performance, reduce private memory usage, and facilitate [assembly trimming](#), which reduces app size.

For more information, see [Reflection versus source generation](#).

Security information

For information about security threats that were considered when designing `JsonSerializer`, and how they can be mitigated, see [System.Text.Json Threat Model](#).

Thread safety

The `System.Text.Json` serializer was designed with thread safety in mind. Practically, this means that once locked, `JsonSerializerOptions` instances can be safely shared across multiple threads. `JsonDocument` provides an immutable, and in .NET 8 and later versions, thread-safe, DOM representation for JSON values.

Additional resources

- [How to use the library](#)

 [Collaborate with us on GitHub](#)

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

 .NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

How to write .NET objects as JSON (serialize)

Article • 10/25/2023

This article shows how to use the [System.Text.Json](#) namespace to serialize to JavaScript Object Notation (JSON). If you're porting existing code from [Newtonsoft.Json](#), see [How to migrate to System.Text.Json](#).

To write JSON to a string or to a file, call the [JsonSerializer.Serialize](#) method.

The following example creates JSON as a string:

C#

```
using System.Text.Json;

namespace SerializeBasic
{
    public class WeatherForecast
    {
        public DateTimeOffset Date { get; set; }
        public int TemperatureCelsius { get; set; }
        public string? Summary { get; set; }
    }

    public class Program
    {
        public static void Main()
        {
            var weatherForecast = new WeatherForecast
            {
                Date = DateTime.Parse("2019-08-01"),
                TemperatureCelsius = 25,
                Summary = "Hot"
            };

            string jsonString = JsonSerializer.Serialize(weatherForecast);

            Console.WriteLine(jsonString);
        }
    }
}

// output:
//{"Date":"2019-08-01T00:00:00-07:00","TemperatureCelsius":25,"Summary":"Hot"}
```

The JSON output is minified (whitespace, indentation, and new-line characters are removed) by default.

The following example uses synchronous code to create a JSON file:

```
C#  
  
using System.Text.Json;  
  
namespace SerializeToFile  
{  
    public class WeatherForecast  
    {  
        public DateTimeOffset Date { get; set; }  
        public int TemperatureCelsius { get; set; }  
        public string? Summary { get; set; }  
    }  
  
    public class Program  
    {  
        public static void Main()  
        {  
            var weatherForecast = new WeatherForecast  
            {  
                Date = DateTime.Parse("2019-08-01"),  
                TemperatureCelsius = 25,  
                Summary = "Hot"  
            };  
  
            string fileName = "WeatherForecast.json";  
            string jsonString = JsonSerializer.Serialize(weatherForecast);  
            File.WriteAllText(fileName, jsonString);  
  
            Console.WriteLine(File.ReadAllText(fileName));  
        }  
    }  
}  
// output:  
//{"Date":"2019-08-01T00:00:00-  
//07:00","TemperatureCelsius":25,"Summary":"Hot"}
```

The following example uses asynchronous code to create a JSON file:

```
C#  
  
using System.Text.Json;  
  
namespace SerializeToFileAsync  
{  
    public class WeatherForecast  
    {  
        public DateTimeOffset Date { get; set; }  
        public int TemperatureCelsius { get; set; }  
        public string? Summary { get; set; }  
    }  
}
```

```

public class Program
{
    public static async Task Main()
    {
        var weatherForecast = new WeatherForecast
        {
            Date = DateTime.Parse("2019-08-01"),
            TemperatureCelsius = 25,
            Summary = "Hot"
        };

        string fileName = "WeatherForecast.json";
        using FileStream createStream = File.Create(fileName);
        await JsonSerializer.SerializeAsync(createStream,
weatherForecast);
        await createStream.DisposeAsync();

        Console.WriteLine(File.ReadAllText(fileName));
    }
}
// output:
//{"Date":"2019-08-01T00:00:00-
07:00","TemperatureCelsius":25,"Summary":"Hot"}

```

The preceding examples use type inference for the type being serialized. An overload of `Serialize()` takes a generic type parameter:

C#

```

using System.Text.Json;

namespace SerializeWithGenericParameter
{
    public class WeatherForecast
    {
        public DateTimeOffset Date { get; set; }
        public int TemperatureCelsius { get; set; }
        public string? Summary { get; set; }
    }

    public class Program
    {
        public static void Main()
        {
            var weatherForecast = new WeatherForecast
            {
                Date = DateTime.Parse("2019-08-01"),
                TemperatureCelsius = 25,
                Summary = "Hot"
            };
        }
    }
}

```

```
        string jsonString = JsonSerializer.Serialize<WeatherForecast>(weatherForecast);

        Console.WriteLine(jsonString);
    }
}
// output:
//{"Date":"2019-08-01T00:00:00-
07:00","TemperatureCelsius":25,"Summary":"Hot"}
```

Serialization behavior

- By default, all public properties are serialized. You can [specify properties to ignore](#). You can also include [private members](#).
- The [default encoder](#) escapes non-ASCII characters, HTML-sensitive characters within the ASCII-range, and characters that must be escaped according to [the RFC 8259 JSON spec ↗](#).
- By default, JSON is minified. You can [pretty-print the JSON](#).
- By default, casing of JSON names matches the .NET names. You can [customize JSON name casing](#).
- By default, circular references are detected and exceptions thrown. You can [preserve references and handle circular references](#).
- By default, [fields](#) are ignored. You can [include fields](#).

When you use `System.Text.Json` indirectly in an ASP.NET Core app, some default behaviors are different. For more information, see [Web defaults for `JsonSerializerOptions`](#).

Supported types include:

- .NET primitives that map to JavaScript primitives, such as numeric types, strings, and Boolean.
- User-defined [plain old CLR objects \(POCOs\)](#).
- One-dimensional and jagged arrays (`T[][]`).
- Collections and dictionaries from the following namespaces:
 - `System.Collections`
 - `System.Collections.Generic`
 - `System.Collections.Immutable`
 - `System.Collections.Concurrent`
 - `System.Collections.Specialized`

- o [System.Collections.ObjectModel](#)

For more information, see [Supported collection types in System.Text.Json](#).

You can [implement custom converters](#) to handle additional types or to provide functionality that isn't supported by the built-in converters.

Here's an example showing how a class that contains collection properties and a user-defined type is serialized:

C#

```
using System.Text.Json;

namespace SerializeExtra
{
    public class WeatherForecast
    {
        public DateTimeOffset Date { get; set; }
        public int TemperatureCelsius { get; set; }
        public string? Summary { get; set; }
        public string? SummaryField;
        public IList<DateTimeOffset>? DatesAvailable { get; set; }
        public Dictionary<string, HighLowTemps>? TemperatureRanges { get;
set; }
        public string[]? SummaryWords { get; set; }
    }

    public class HighLowTemps
    {
        public int High { get; set; }
        public int Low { get; set; }
    }

    public class Program
    {
        public static void Main()
        {
            var weatherForecast = new WeatherForecast
            {
                Date = DateTime.Parse("2019-08-01"),
                TemperatureCelsius = 25,
                Summary = "Hot",
                SummaryField = "Hot",
                DatesAvailable = new List<DateTimeOffset>()
                    { DateTime.Parse("2019-08-01"), DateTime.Parse("2019-08-
02") },
                TemperatureRanges = new Dictionary<string, HighLowTemps>
                {
                    ["Cold"] = new HighLowTemps { High = 20, Low = -10
},
                    ["Hot"] = new HighLowTemps { High = 60 , Low = 20 }
                },
            };
        }
    }
}
```

```

        SummaryWords = new[] { "Cool", "Windy", "Humid" }
    };

        var options = new JsonSerializerOptions { WriteIndented = true
    };
        string jsonString = JsonSerializer.Serialize(weatherForecast,
options);

        Console.WriteLine(jsonString);
    }
}
// output:
//{
//  "Date": "2019-08-01T00:00:00-07:00",
//  "TemperatureCelsius": 25,
//  "Summary": "Hot",
//  "DatesAvailable": [
//    "2019-08-01T00:00:00-07:00",
//    "2019-08-02T00:00:00-07:00"
//  ],
//  "TemperatureRanges": {
//    "Cold": {
//      "High": 20,
//      "Low": -10
//    },
//    "Hot": {
//      "High": 60,
//      "Low": 20
//    }
//  },
//  "SummaryWords": [
//    "Cool",
//    "Windy",
//    "Humid"
//  ]
//}

```

Serialize to UTF-8

It's 5-10% faster to serialize to a UTF-8 byte array than to use the string-based methods. That's because the bytes (as UTF-8) don't need to be converted to strings (UTF-16).

To serialize to a UTF-8 byte array, call the [JsonSerializer.SerializeToUtf8Bytes](#) method:

C#

```
byte[] jsonUtf8Bytes = JsonSerializer.SerializeToUtf8Bytes(weatherForecast);
```

A [Serialize](#) overload that takes a [Utf8JsonWriter](#) is also available.

Serialize to formatted JSON

To pretty-print the JSON output, set `JsonSerializerOptions.WriteIndented` to `true`:

C#

```
using System.Text.Json;

namespace SerializeWriteIndented
{
    public class WeatherForecast
    {
        public DateTimeOffset Date { get; set; }
        public int TemperatureCelsius { get; set; }
        public string? Summary { get; set; }
    }

    public class Program
    {
        public static void Main()
        {
            var weatherForecast = new WeatherForecast
            {
                Date = DateTime.Parse("2019-08-01"),
                TemperatureCelsius = 25,
                Summary = "Hot"
            };

            var options = new JsonSerializerOptions { WriteIndented = true };
            string jsonString = JsonSerializer.Serialize(weatherForecast,
options);

            Console.WriteLine(jsonString);
        }
    }
}

// output:
//{
//    "Date": "2019-08-01T00:00:00-07:00",
//    "TemperatureCelsius": 25,
//    "Summary": "Hot"
//}
```

💡 Tip

If you use `JsonSerializerOptions` repeatedly with the same options, don't create a new `JsonSerializerOptions` instance each time you use it. Reuse the same instance for every call. For more information, see [Reuse JsonSerializerOptions instances](#).

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

.NET is an open source project.
Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

How to customize property names and values with System.Text.Json

Article • 10/18/2023

By default, property names and dictionary keys are unchanged in the JSON output, including case. Enum values are represented as numbers. And properties are serialized in the order they're defined. However, you can customize these behaviors by:

- Specifying specific serialized property names.
- Using a built-in [naming policy](#), such as camelCase, snake_case, or kebab-case, for property names and dictionary keys.
- Using a custom naming policy for property names and dictionary keys.
- Serializing enum values as strings, with or without a naming policy.
- Configuring the order of serialized properties.

ⓘ Note

The [web default](#) naming policy is camel case.

For other scenarios that require special handling of JSON property names and values, you can [implement custom converters](#).

Customize individual property names

To set the name of individual properties, use the [\[JsonPropertyName\]](#) attribute.

Here's an example type to serialize and resulting JSON:

C#

```
public class WeatherForecastWithPropertyNameAttribute
{
    public DateTimeOffset Date { get; set; }
    public int TemperatureCelsius { get; set; }
    public string? Summary { get; set; }
    [JsonPropertyName("Wind")]
    public int WindSpeed { get; set; }
}
```

JSON

```
{
    "Date": "2019-08-01T00:00:00-07:00",
    "TemperatureCelsius": 25,
    "Summary": "Hot",
    "Wind": 35
}
```

The property name set by this attribute:

- Applies in both directions, for serialization and deserialization.
- Takes precedence over property naming policies.
- Doesn't affect parameter name matching for parameterized constructors.

Use a built-in naming policy

The following table shows the built-in naming policies and how they affect property names.

| Naming policy | Description | Original property name | Converted property name |
|-----------------|---|------------------------|-------------------------|
| CamelCase | First word starts with a lower case character. Successive words start with an uppercase character. | TempCelsius | tempCelsius |
| KebabCaseLower* | Words are separated by hyphens. All characters are lowercase. | TempCelsius | temp-celsius |
| KebabCaseUpper* | Words are separated by hyphens. All characters are uppercase. | TempCelsius | TEMP-CELSIUS |
| SnakeCaseLower* | Words are separated by underscores. All characters are lowercase. | TempCelsius | temp_celsius |
| SnakeCaseUpper* | Words are separated by underscores. All characters are uppercase. | TempCelsius | TEMP_CELSIUS |

* Available in .NET 8 and later versions.

The following example shows how to use camel case for all JSON property names by setting `JsonSerializerOptions.PropertyNamingPolicy` to `JsonNamingPolicy.CamelCase`:

C#

```
var serializeOptions = new JsonSerializerOptions
{
    PropertyNamingPolicy = JsonNamingPolicy.CamelCase,
    WriteIndented = true
};
jsonString = JsonSerializer.Serialize(weatherForecast, serializeOptions);
```

Here's an example class to serialize and JSON output:

C#

```
public class WeatherForecastWithPropertyNameAttribute
{
    public DateTimeOffset Date { get; set; }
    public int TemperatureCelsius { get; set; }
    public string? Summary { get; set; }
    [JsonPropertyName("Wind")]
    public int WindSpeed { get; set; }
}
```

JSON

```
{
    "date": "2019-08-01T00:00:00-07:00",
    "temperatureCelsius": 25,
    "summary": "Hot",
    "Wind": 35
}
```

The naming policy:

- Applies to serialization and deserialization.
- Is overridden by `[JsonPropertyName]` attributes. This is why the JSON property name `Wind` in the example is not camel case.

ⓘ Note

None of the built-in naming policies support letters that are surrogate pairs. For more information, see [dotnet/runtime issue 90352](#).

Use a custom JSON property naming policy

To use a custom JSON property naming policy, create a class that derives from [JsonNamingPolicy](#) and override the [ConvertName](#) method, as shown in the following example:

C#

```
using System.Text.Json;

namespace SystemTextJsonSamples
{
    public class UpperCaseNamingPolicy : JsonNamingPolicy
    {
        public override string ConvertName(string name) =>
            name.ToUpper();
    }
}
```

Then set the [JsonSerializerOptions.PropertyNamingPolicy](#) property to an instance of your naming policy class:

C#

```
var options = new JsonSerializerOptions
{
    PropertyNamingPolicy = new UpperCaseNamingPolicy(),
    WriteIndented = true
};
jsonString = JsonSerializer.Serialize(weatherForecast, options);
```

Here's an example class to serialize and JSON output:

C#

```
public class WeatherForecastWithPropertyNameAttribute
{
    public DateTimeOffset Date { get; set; }
    public int TemperatureCelsius { get; set; }
    public string? Summary { get; set; }
    [JsonPropertyName("Wind")]
    public int WindSpeed { get; set; }
}
```

JSON

```
{
    "DATE": "2019-08-01T00:00:00-07:00",
    "TEMPERATURECELSIUS": 25,
    "SUMMARY": "Hot",
```

```
    "Wind": 35  
}
```

The JSON property naming policy:

- Applies to serialization and deserialization.
- Is overridden by `[JsonPropertyName]` attributes. This is why the JSON property name `Wind` in the example is not upper case.

Use a naming policy for dictionary keys

If a property of an object to be serialized is of type `Dictionary<string, TValue>`, the `string` keys can be converted using a naming policy, such as camel case. To do that, set `JsonSerializerOptions.DictionaryKeyPolicy` to your desired naming policy. The following example uses the `CamelCase` naming policy:

C#

```
var options = new JsonSerializerOptions  
{  
    DictionaryKeyPolicy = JsonNamingPolicy.CamelCase,  
    WriteIndented = true  
};  
jsonString = JsonSerializer.Serialize(weatherForecast, options);
```

Serializing an object with a dictionary named `TemperatureRanges` that has key-value pairs `"ColdMinTemp", 20` and `"HotMinTemp", 40` would result in JSON output like the following example:

JSON

```
{  
    "Date": "2019-08-01T00:00:00-07:00",  
    "TemperatureCelsius": 25,  
    "Summary": "Hot",  
    "TemperatureRanges": {  
        "coldMinTemp": 20,  
        "hotMinTemp": 40  
    }  
}
```

Naming policies for dictionary keys apply to serialization only. If you deserialize a dictionary, the keys will match the JSON file even if you set `JsonSerializerOptions.DictionaryKeyPolicy` to a non-default naming policy.

Enums as strings

By default, enums are serialized as numbers. To serialize enum names as strings, use the [JsonStringEnumConverter](#) or [JsonStringEnumConverter<TEnum>](#) converter. Only [JsonStringEnumConverter<TEnum>](#) is supported by the Native AOT runtime.

For example, suppose you need to serialize the following class that has an enum:

C#

```
public class WeatherForecastWithEnum
{
    public DateTimeOffset Date { get; set; }
    public int TemperatureCelsius { get; set; }
    public Summary? Summary { get; set; }
}

public enum Summary
{
    Cold, Cool, Warm, Hot
}
```

If the Summary is `Hot`, by default the serialized JSON has the numeric value 3:

JSON

```
{
    "Date": "2019-08-01T00:00:00-07:00",
    "TemperatureCelsius": 25,
    "Summary": 3
}
```

The following sample code serializes the enum names instead of the numeric values, and converts the names to camel case:

C#

```
options = new JsonSerializerOptions
{
    WriteIndented = true,
    Converters =
    {
        new JsonStringEnumConverter(JsonNamingPolicy.CamelCase)
    }
};
jsonString = JsonSerializer.Serialize(weatherForecast, options);
```

The resulting JSON looks like the following example:

JSON

```
{  
    "Date": "2019-08-01T00:00:00-07:00",  
    "TemperatureCelsius": 25,  
    "Summary": "hot"  
}
```

The built-in [JsonStringEnumConverter](#) can deserialize string values as well. It works with or without a specified naming policy. The following example shows deserialization using [CamelCase](#):

C#

```
options = new JsonSerializerOptions  
{  
    Converters =  
    {  
        new JsonStringEnumConverter(JsonNamingPolicy.CamelCase)  
    }  
};  
weatherForecast = JsonSerializer.Deserialize<WeatherForecastWithEnum>  
(jsonString, options);
```

You can also specify the converter to use by annotating your enum with [JsonConverterAttribute](#). The following example shows how to specify the [JsonStringEnumConverter<TEnum>](#) (available in .NET 8 and later versions) by using the [JsonConverterAttribute](#) attribute. For example, suppose you need to serialize the following class that has an enum:

C#

```
public class WeatherForecastWithPrecipEnum  
{  
    public DateTimeOffset Date { get; set; }  
    public int TemperatureCelsius { get; set; }  
    public Precipitation? Precipitation { get; set; }  
  
    [JsonConverter(typeof(JsonStringEnumConverter<Precipitation>))]  
    public enum Precipitation  
    {  
        Drizzle, Rain, Sleet, Hail, Snow  
    }  
}
```

The following sample code serializes the enum names instead of the numeric values:

C#

```
var options = new JsonSerializerOptions
{
    WriteIndented = true,
};
jsonString = JsonSerializer.Serialize(weatherForecast, options);
```

The resulting JSON looks like the following example:

JSON

```
{
    "Date": "2019-08-01T00:00:00-07:00",
    "TemperatureCelsius": 25,
    "Precipitation": "Sleet"
}
```

To use the converter with source generation, see [Serialize enum fields as strings](#).

Configure the order of serialized properties

By default, properties are serialized in the order in which they're defined in their class. The [\[JsonPropertyOrder\]](#) attribute lets you specify the order of properties in the JSON output from serialization. The default value of the `Order` property is zero. Set `Order` to a positive number to position a property after those that have the default value. A negative `Order` positions a property before those that have the default value. Properties are written in order from the lowest `Order` value to the highest. Here's an example:

C#

```
using System.Text.Json;
using System.Text.Json.Serialization;

namespace PropertyOrder
{
    public class WeatherForecast
    {
        [JsonPropertyOrder(-5)]
        public DateTime Date { get; set; }
        public int TemperatureC { get; set; }
        [JsonPropertyOrder(-2)]
        public int TemperatureF { get; set; }
        [JsonPropertyOrder(5)]
        public string? Summary { get; set; }
        [JsonPropertyOrder(2)]
        public int WindSpeed { get; set; }
    }
}
```

```
}

public class Program
{
    public static void Main()
    {
        var weatherForecast = new WeatherForecast
        {
            Date = DateTime.Parse("2019-08-01"),
            TemperatureC = 25,
            TemperatureF = 25,
            Summary = "Hot",
            WindSpeed = 10
        };

        var options = new JsonSerializerOptions { WriteIndented = true };
        string jsonString = JsonSerializer.Serialize(weatherForecast,
options);
        Console.WriteLine(jsonString);
    }
}
// output:
//{
//    "Date": "2019-08-01T00:00:00",
//    "TemperatureF": 25,
//    "TemperatureC": 25,
//    "WindSpeed": 10,
//    "Summary": "Hot"
//}
```

See also

- [System.Text.Json overview](#)
- [How to serialize and deserialize JSON](#)

 Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback [here](#).

 [Open a documentation issue](#)

 [Provide product feedback](#)

How to ignore properties with System.Text.Json

Article • 05/26/2023

When serializing C# objects to JavaScript Object Notation (JSON), by default, all public properties are serialized. If you don't want some of them to appear in the resulting JSON, you have several options. In this article, you learn how to ignore properties based on various criteria:

- Individual properties
- All read-only properties
- All null-value properties
- All default-value properties

Ignore individual properties

To ignore individual properties, use the `[JsonIgnore]` attribute.

The following example shows a type to serialize. It also shows the JSON output:

C#

```
public class WeatherForecastWithIgnoreAttribute
{
    public DateTimeOffset Date { get; set; }
    public int TemperatureCelsius { get; set; }
    [JsonIgnore]
    public string? Summary { get; set; }
}
```

JSON

```
{
    "Date": "2019-08-01T00:00:00-07:00",
    "TemperatureCelsius": 25,
}
```

You can specify conditional exclusion by setting the `[JsonIgnore]` attribute's `Condition` property. The `JsonIgnoreCondition` enum provides the following options:

- `Always` - The property is always ignored. If no `Condition` is specified, this option is assumed.

- `Never` - The property is always serialized and deserialized, regardless of the `DefaultIgnoreCondition`, `IgnoreReadOnlyProperties`, and `IgnoreReadOnlyFields` global settings.
- `WhenWritingDefault` - The property is ignored on serialization if it's a reference type `null`, a nullable value type `null`, or a value type `default`.
- `WhenWritingNull` - The property is ignored on serialization if it's a reference type `null`, or a nullable value type `null`.

The following example illustrates the use of the `[JsonIgnore]` attribute's `Condition` property:

C#

```
using System.Text.Json;
using System.Text.Json.Serialization;

namespace JsonIgnoreAttributeExample
{
    public class Forecast
    {
        [JsonIgnore(Condition = JsonIgnoreCondition.WhenWritingDefault)]
        public DateTime Date { get; set; }

        [JsonIgnore(Condition = JsonIgnoreCondition.Never)]
        public int TemperatureC { get; set; }

        [JsonIgnore(Condition = JsonIgnoreCondition.WhenWritingNull)]
        public string? Summary { get; set; }
    };

    public class Program
    {
        public static void Main()
        {
            Forecast forecast = new()
            {
                Date = default,
                Summary = null,
                TemperatureC = default
            };

            JsonSerializerOptions options = new()
            {
                DefaultIgnoreCondition =
JsonIgnoreCondition.WhenWritingDefault
            };

            string forecastJson =
JsonSerializer.Serialize<Forecast>(forecast,options);

            Console.WriteLine(forecastJson);
        }
    }
}
```

```
        }
    }

// Produces output like the following example:
//
//>{"TemperatureC":0}
```

Ignore all read-only properties

A property is read-only if it contains a public getter but not a public setter. To ignore all read-only properties when serializing, set the [JsonSerializerOptions.IgnoreReadOnlyProperties](#) to `true`, as shown in the following example:

C#

```
var options = new JsonSerializerOptions
{
    IgnoreReadOnlyProperties = true,
    WriteIndented = true
};
jsonString = JsonSerializer.Serialize(weatherForecast, options);
```

The following example shows a type to serialize. It also shows the JSON output:

C#

```
public class WeatherForecastWithR0Property
{
    public DateTimeOffset Date { get; set; }
    public int TemperatureCelsius { get; set; }
    public string? Summary { get; set; }
    public int WindSpeedReadOnly { get; private set; } = 35;
}
```

JSON

```
{
    "Date": "2019-08-01T00:00:00-07:00",
    "TemperatureCelsius": 25,
    "Summary": "Hot",
}
```

This option applies only to properties. To ignore read-only fields when [serializing fields](#), use the [JsonSerializerOptions.IgnoreReadOnlyFields](#) global setting.

Ignore all null-value properties

To ignore all null-value properties, set the [DefaultIgnoreCondition](#) property to [WhenWritingNull](#), as shown in the following example:

C#

```
using System.Text.Json;
using System.Text.Json.Serialization;

namespace IgnoreNullOnSerialize
{
    public class Forecast
    {
        public DateTime Date { get; set; }
        public int TemperatureC { get; set; }
        public string? Summary { get; set; }
    };

    public class Program
    {
        public static void Main()
        {
            Forecast forecast = new()
            {
                Date = DateTime.Now,
                Summary = null,
                TemperatureC = default
            };

            JsonSerializerOptions options = new()
            {
                DefaultIgnoreCondition = JsonIgnoreCondition.WhenWritingNull
            };

            string forecastJson =
                JsonSerializer.Serialize<Forecast>(forecast, options);

            Console.WriteLine(forecastJson);
        }
    }
}

// Produces output like the following example:
// [{"Date": "2020-10-30T10:11:40.2359135-07:00", "TemperatureC": 0}
```

Ignore all default-value properties

To prevent serialization of default values in value type properties, set the [DefaultIgnoreCondition](#) property to [WhenWritingDefault](#), as shown in the following example:

C#

```
using System.Text.Json;
using System.Text.Json.Serialization;

namespace IgnoreValueDefaultOnSerialize
{
    public class Forecast
    {
        public DateTime Date { get; set; }
        public int TemperatureC { get; set; }
        public string? Summary { get; set; }
    };

    public class Program
    {
        public static void Main()
        {
            Forecast forecast = new()
            {
                Date = DateTime.Now,
                Summary = null,
                TemperatureC = default
            };

            JsonSerializerOptions options = new()
            {
                DefaultIgnoreCondition =
JsonIgnoreCondition.WhenWritingDefault
            };

            string forecastJson =
                JsonSerializer.Serialize<Forecast>(forecast, options);

            Console.WriteLine(forecastJson);
        }
    }
}

// Produces output like the following example:
// 
// {"Date":"2020-10-21T15:40:06.8920138-07:00"}  

```

The [WhenWritingDefault](#) setting also prevents serialization of null-value reference type and nullable value type properties.

See also

- [System.Text.Json overview](#)
- [How to serialize and deserialize JSON](#)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

.NET is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

Include fields

Article • 10/25/2023

By default, fields aren't serialized. Use the `JsonSerializerOptions.IncludeFields` global setting or the `[JsonInclude]` attribute to include fields when serializing or deserializing, as shown in the following example:

C#

```
using System.Text.Json;
using System.Text.Json.Serialization;

namespace Fields
{
    public class Forecast
    {
        public DateTime Date;
        public int TemperatureC;
        public string? Summary;
    }

    public class Forecast2
    {
        [JsonInclude]
        public DateTime Date;
        [JsonInclude]
        public int TemperatureC;
        [JsonInclude]
        public string? Summary;
    }

    public class Program
    {
        public static void Main()
        {
            var json =
                @"{""Date"":""2020-09-
06T11:31:01.923395"",""TemperatureC"":-1,""Summary"":""Cold""} ";
            Console.WriteLine($"Input JSON: {json}");

            var options = new JsonSerializerOptions
            {
                IncludeFields = true,
            };
            var forecast = JsonSerializer.Deserialize<Forecast>(json,
options)!;

            Console.WriteLine($"forecast.Date: {forecast.Date}");
            Console.WriteLine($"forecast.TemperatureC:
{forecast.TemperatureC}");
            Console.WriteLine($"forecast.Summary: {forecast.Summary}");
        }
    }
}
```

```

        var roundTrippedJson =
            JsonSerializer.Serialize<Forecast>(forecast, options);

        Console.WriteLine($"Output JSON: {roundTrippedJson}");

        var forecast2 = JsonSerializer.Deserialize<Forecast2>(json)!;

        Console.WriteLine($"forecast2.Date: {forecast2.Date}");
        Console.WriteLine($"forecast2.TemperatureC:
{forecast2.TemperatureC}");
        Console.WriteLine($"forecast2.Summary: {forecast2.Summary}");

        roundTrippedJson = JsonSerializer.Serialize<Forecast2>
(forecast2);

        Console.WriteLine($"Output JSON: {roundTrippedJson}");
    }
}
}

// Produces output like the following example:
//
//Input JSON: { "Date":"2020-09-
06T11:31:01.923395", "TemperatureC": -1, "Summary": "Cold" }
//forecast.Date: 9/6/2020 11:31:01 AM
//forecast.TemperatureC: -1
//forecast.Summary: Cold
//Output JSON: { "Date": "2020-09-
06T11:31:01.923395", "TemperatureC": -1, "Summary": "Cold" }
//forecast2.Date: 9/6/2020 11:31:01 AM
//forecast2.TemperatureC: -1
//forecast2.Summary: Cold
//Output JSON: { "Date": "2020-09-
06T11:31:01.923395", "TemperatureC": -1, "Summary": "Cold" }

```

To ignore read-only fields, use the [JsonSerializerOptions.IgnoreReadOnlyFields](#) global setting.

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

How to read JSON as .NET objects (deserialize)

Article • 10/25/2023

This article shows how to use the [System.Text.Json](#) namespace to serialize to and deserialize from JavaScript Object Notation (JSON). If you're porting existing code from [Newtonsoft.Json](#), see [How to migrate to System.Text.Json](#).

A common way to deserialize JSON is to have (or create) a .NET class with properties and fields that represent one or more of the JSON properties. Then, to deserialize from a string or a file, call the [JsonSerializer.Deserialize](#) method. For the generic overloads, the generic type parameter is the .NET class. For the non-generic overloads, you pass the type of the class as a method parameter. You can deserialize either synchronously or asynchronously.

Any JSON properties that aren't represented in your class are ignored [by default](#). Also, if any properties on the type are [required](#) but not present in the JSON payload, deserialization will fail.

The following example shows how to deserialize a JSON string:

C#

```
using System.Text.Json;

namespace DeserializeExtra
{
    public class WeatherForecast
    {
        public DateTimeOffset Date { get; set; }
        public int TemperatureCelsius { get; set; }
        public string? Summary { get; set; }
        public string? SummaryField;
        public IList<DateTimeOffset>? DatesAvailable { get; set; }
        public Dictionary<string, HighLowTemps>? TemperatureRanges { get;
set; }
        public string[]? SummaryWords { get; set; }
    }

    public class HighLowTemps
    {
        public int High { get; set; }
        public int Low { get; set; }
    }

    public class Program
```

```

{
    public static void Main()
    {
        string jsonString =
@"{
    ""Date"": ""2019-08-01T00:00:00-07:00"",
    ""TemperatureCelsius"": 25,
    ""Summary"": ""Hot"",
    ""DatesAvailable"": [
        ""2019-08-01T00:00:00-07:00"",
        ""2019-08-02T00:00:00-07:00""
    ],
    ""TemperatureRanges"": {
        ""Cold"": {
            ""High"": 20,
            ""Low"": -10
        },
        ""Hot"": {
            ""High"": 60,
            ""Low"": 20
        }
    },
    ""SummaryWords"": [
        ""Cool"",
        ""Windy"",
        ""Humid""
    ]
}
";

```

```

        WeatherForecast? weatherForecast =
            JsonSerializer.Deserialize<WeatherForecast>(jsonString);

        Console.WriteLine($"Date: {weatherForecast?.Date}");
        Console.WriteLine($"TemperatureCelsius:
{weatherForecast?.TemperatureCelsius}");
        Console.WriteLine($"Summary: {weatherForecast?.Summary}");
    }
}
// output:
//Date: 8/1/2019 12:00:00 AM -07:00
//TemperatureCelsius: 25
//Summary: Hot

```

To deserialize from a file by using synchronous code, read the file into a string, as shown in the following example:

```
C#
using System.Text.Json;
namespace DeserializeFromFile
```

```

{
    public class WeatherForecast
    {
        public DateTimeOffset Date { get; set; }
        public int TemperatureCelsius { get; set; }
        public string? Summary { get; set; }
    }

    public class Program
    {
        public static void Main()
        {
            string fileName = "WeatherForecast.json";
            string jsonString = File.ReadAllText(fileName);
            WeatherForecast weatherForecast =
JsonSerializer.Deserialize<WeatherForecast>(jsonString)!

            Console.WriteLine($"Date: {weatherForecast.Date}");
            Console.WriteLine($"TemperatureCelsius:
{weatherForecast.TemperatureCelsius}");
            Console.WriteLine($"Summary: {weatherForecast.Summary}");
        }
    }
}

// output:
//Date: 8/1/2019 12:00:00 AM -07:00
//TemperatureCelsius: 25
//Summary: Hot

```

To deserialize from a file by using asynchronous code, call the [DeserializeAsync](#) method:

```

C#

using System.Text.Json;

namespace DeserializeFromFileAsync
{
    public class WeatherForecast
    {
        public DateTimeOffset Date { get; set; }
        public int TemperatureCelsius { get; set; }
        public string? Summary { get; set; }
    }

    public class Program
    {
        public static async Task Main()
        {
            string fileName = "WeatherForecast.json";
            using FileStream openStream = File.OpenRead(fileName);
            WeatherForecast? weatherForecast =
                await JsonSerializer.DeserializeAsync<WeatherForecast>
(openStream);

```

```
        Console.WriteLine($"Date: {weatherForecast?.Date}");
        Console.WriteLine($"TemperatureCelsius:
{weatherForecast?.TemperatureCelsius}");
        Console.WriteLine($"Summary: {weatherForecast?.Summary}");
    }
}
// output:
//Date: 8/1/2019 12:00:00 AM -07:00
//TemperatureCelsius: 25
//Summary: Hot
```

Deserialization behavior

The following behaviors apply when deserializing JSON:

- By default, property name matching is case-sensitive. You can [specify case-insensitivity](#).
- Non-public constructors are ignored by the serializer.
- Deserialization to immutable objects or properties that don't have public `set` accessors is supported but not enabled by default. See [Immutable types and records](#).
- By default, enums are supported as numbers. You can [deserialize string enum fields](#).
- By default, fields are ignored. You can [include fields](#).
- By default, comments or trailing commas in the JSON throw exceptions. You can [allow comments and trailing commas](#).
- The [default maximum depth](#) is 64.

When you use `System.Text.Json` indirectly in an ASP.NET Core app, some default behaviors are different. For more information, see [Web defaults for `JsonSerializerOptions`](#).

You can [implement custom converters](#) to provide functionality that isn't supported by the built-in converters.

Deserialize without a .NET class

If you have JSON that you want to deserialize, and you don't have the class to deserialize it into, you have options other than manually creating the class that you need:

- Use the [Utf8JsonReader](#) directly.

- Deserialize into a [JSON DOM \(document object model\)](#) and extract what you need from the DOM.

The DOM lets you navigate to a subsection of a JSON payload and deserialize a single value, a custom type, or an array. For information about the [JsonNode](#) DOM, see [Deserialize subsections of a JSON payload](#). For information about the [JsonDocument](#) DOM, see [How to search a JsonDocument and JsonElement for sub-elements](#).

- Use Visual Studio 2022 to automatically generate the class you need:
 - Copy the JSON that you need to deserialize.
 - Create a class file and delete the template code.
 - Choose **Edit > Paste Special > Paste JSON as Classes**.

The result is a class that you can use for your deserialization target.

Deserialize from UTF-8

To deserialize from UTF-8, call a [JsonSerializer.Deserialize](#) overload that takes a `ReadOnlySpan<byte>` or a `Utf8JsonReader`, as shown in the following examples. The examples assume the JSON is in a byte array named `jsonUtf8Bytes`.

C#

```
var readOnlySpan = new ReadOnlySpan<byte>(jsonUtf8Bytes);
WeatherForecast serializedWeatherForecast =
    JsonSerializer.Deserialize<WeatherForecast>(readOnlySpan);
```

C#

```
var utf8Reader = new Utf8JsonReader(jsonUtf8Bytes);
WeatherForecast serializedWeatherForecast =
    JsonSerializer.Deserialize<WeatherForecast>(ref utf8Reader);
```

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For

.NET

.NET feedback

.NET is an open source project.
Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

more information, see [our contributor guide](#).

Required properties

Article • 10/04/2022

Starting in .NET 7, you can mark certain properties to signify that they must be present in the JSON payload for deserialization to succeed. If one or more of these required properties is not present, the [JsonSerializer.Deserialize](#) methods throw a [JsonException](#).

There are three ways to mark a property or field as required for JSON deserialization:

- By adding the [required modifier](#), which is new in C# 11.
- By annotating it with [JsonRequiredAttribute](#), which is new in .NET 7.
- By modifying the [JsonPropertyInfo.IsRequired](#) property of the contract model, which is new in .NET 7.

From the serializer's perspective, these two demarcations are equivalent and both map to the same piece of metadata, which is [JsonPropertyInfo.IsRequired](#). In most cases, you'd simply use the built-in C# keyword. However, in the following cases, you should use [JsonRequiredAttribute](#) instead:

- If you're using a programming language other than C# or a down-level version of C#.
- If you only want the requirement to apply to JSON deserialization.
- If you're using `System.Text.Json` serialization in [source generation](#) mode. In this case, your code won't compile if you use the `required` modifier, as source generation occurs at compile time.

The following code snippet shows an example of a property modified with the `required` keyword. This property must be present in the JSON payload for deserialization to succeed.

C#

```
using System.Text.Json;

// The following line throws a JsonException at run time.
Console.WriteLine(JsonSerializer.Deserialize<Person>("""{"Age": 42}"""));

public class Person
{
    public required string Name { get; set; }
    public int Age { get; set; }
}
```

Alternatively, you can use [JsonRequiredAttribute](#):

C#

```
using System.Text.Json;

// The following line throws a JsonException at run time.
Console.WriteLine(JsonSerializer.Deserialize<Person>("""{"Age": 42}"""));

public class Person
{
    [JsonPropertyRequired]
    public string Name { get; set; }
    public int Age { get; set; }
}
```

It's also possible to control whether a property is required via the contract model using the `JsonPropertyInfo.isRequired` property:

C#

```
var options = new JsonSerializerOptions
{
    TypeInfoResolver = new DefaultJsonTypeInfoResolver
    {
        Modifiers =
        {
            static TypeInfo =>
            {
                if (typeInfo.Kind != JsonTypeInfoKind.Object)
                    return;

                foreach (JsonPropertyInfo propertyInfo in
typeInfo.Properties)
                {
                    // Strip IsRequired constraint from every property.
                    propertyInfo.isRequired = false;
                }
            }
        }
    };
};

// Deserialization now succeeds even though the Name property isn't in the
// JSON payload.
JsonSerializer.Deserialize<Person>("""{"Age": 42}""", options);
```

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

How to allow some kinds of invalid JSON with System.Text.Json

Article • 05/26/2023

In this article, you will learn how to allow comments, trailing commas, and quoted numbers in JSON, and how to write numbers as strings.

Allow comments and trailing commas

By default, comments and trailing commas are not allowed in JSON. To allow comments in the JSON, set the `JsonSerializerOptions.ReadCommentHandling` property to `JsonCommentHandling.Skip`. And to allow trailing commas, set the `JsonSerializerOptions.AllowTrailingCommas` property to `true`. The following example shows how to allow both:

C#

```
var options = new JsonSerializerOptions
{
    ReadCommentHandling = JsonCommentHandling.Skip,
    AllowTrailingCommas = true,
};
var weatherForecast = JsonSerializer.Deserialize<WeatherForecast>
(jsonString, options);
```

Here's example JSON with comments and a trailing comma:

JSON

```
{
    "Date": "2019-08-01T00:00:00-07:00",
    "TemperatureCelsius": 25, // Fahrenheit 77
    "Summary": "Hot", /* Zharko */
    // Comments on
    /* separate lines */
}
```

Allow or write numbers in quotes

Some serializers encode numbers as JSON strings (surrounded by quotes).

For example:

JSON

```
{  
    "DegreesCelsius": "23"  
}
```

Instead of:

JSON

```
{  
    "DegreesCelsius": 23  
}
```

To serialize numbers in quotes or accept numbers in quotes across the entire input object graph, set [JsonSerializerOptions.NumberHandling](#) as shown in the following example:

C#

```
using System.Text.Json;  
using System.Text.Json.Serialization;  
  
namespace QuotedNumbers  
{  
    public class Forecast  
    {  
        public DateTime Date { get; init; }  
        public int TemperatureC { get; set; }  
        public string? Summary { get; set; }  
    };  
  
    public class Program  
    {  
        public static void Main()  
        {  
            Forecast forecast = new()  
            {  
                Date = DateTime.Now,  
                TemperatureC = 40,  
                Summary = "Hot"  
            };  
  
            JsonSerializerOptions options = new()  
            {  
                NumberHandling =  
                    JsonNumberHandling.AllowReadingFromString |  
                    JsonNumberHandling.WriteString,  
                WriteIndented = true  
            };
```

```

        string forecastJson =
            JsonSerializer.Serialize<Forecast>(forecast, options);

        Console.WriteLine($"Output JSON:\n{forecastJson}");

        Forecast forecastDeserialized =
            JsonSerializer.Deserialize<Forecast>(forecastJson,
options)!;

        Console.WriteLine($"Date: {forecastDeserialized.Date}");
        Console.WriteLine($"TemperatureC:
{forecastDeserialized.TemperatureC}");
        Console.WriteLine($"Summary: {forecastDeserialized.Summary}");
    }
}

// Produces output like the following example:
//
//Output JSON:
//{
//  "Date": "2020-10-23T12:27:06.4017385-07:00",
//  "TemperatureC": "40",
//  "Summary": "Hot"
//}
//Date: 10/23/2020 12:27:06 PM
//TemperatureC: 40
//Summary: Hot

```

When you use `System.Text.Json` indirectly through ASP.NET Core, quoted numbers are allowed when deserializing because ASP.NET Core specifies [web default options](#).

To allow or write quoted numbers for specific properties, fields, or types, use the [\[JsonNumberHandling\]](#) attribute.

See also

- [System.Text.Json overview](#)
- [How to serialize and deserialize JSON](#)

 Collaborate with us on
GitHub

The source for this content can
be found on GitHub, where you
can also create and review



.NET feedback

.NET is an open source project.
Select a link to provide feedback:

 [Open a documentation issue](#)

issues and pull requests. For more information, see [our contributor guide](#).

 [Provide product feedback](#)

Handle missing members during deserialization

Article • 03/03/2023

By default, if the JSON payload you're deserializing contains properties that don't exist in the deserialized plain old CLR object (POCO) type, they're simply ignored. Starting in .NET 8, you can specify that all members must be present in the payload. If they're not, a [JsonException](#) exception is thrown. You can configure this behavior in one of three ways:

- Annotate your POCO type with the

```
[<xref:System.Text.Json.Serialization.JsonUnmappedMemberHandlingAttribute>]  
attribute, specifying either to Skip or Disallow missing members.
```

C#

```
[JsonUnmappedMemberHandling(JsonUnmappedMemberHandling.Disallow)]  
public class MyPoco  
{  
    public int Id { get; set; }  
}  
  
JsonSerializer.Deserialize<MyPoco>(""""{"Id" : 42, "AnotherId" : -1  
}""");  
// JsonException : The JSON property 'AnotherId' could not be mapped to  
any .NET member contained in type 'MyPoco'.
```

- Set [JsonSerializerOptions.UnmappedMemberHandling](#) to either [Skip](#) or [Disallow](#).
- Customize the [JsonTypeInfo](#) contract for the relevant type. (For information about customizing a contract, see [Customize a JSON contract](#).)

 Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

How to handle overflow JSON or use JsonElement or JsonNode in System.Text.Json

Article • 05/26/2023

This article shows how to handle overflow JSON with the `System.Text.Json` namespace. It also shows how to deserialize into `JsonElement` or `JsonNode`, as an alternative for other scenarios where the target type might not perfectly match all of the JSON being deserialized.

Handle overflow JSON

While deserializing, you might receive data in the JSON that is not represented by properties of the target type. For example, suppose your target type is this:

```
C#  
  
public class WeatherForecast  
{  
    public DateTimeOffset Date { get; set; }  
    public int TemperatureCelsius { get; set; }  
    public string? Summary { get; set; }  
}
```

And the JSON to be deserialized is this:

```
JSON  
  
{  
    "Date": "2019-08-01T00:00:00-07:00",  
    "temperatureCelsius": 25,  
    "Summary": "Hot",  
    "DatesAvailable": [  
        "2019-08-01T00:00:00-07:00",  
        "2019-08-02T00:00:00-07:00"  
    ],  
    "SummaryWords": [  
        "Cool",  
        "Windy",  
        "Humid"  
    ]  
}
```

If you deserialize the JSON shown into the type shown, the `DatesAvailable` and `SummaryWords` properties have nowhere to go and are lost. To capture extra data such as these properties, apply the `[JsonExtensionData]` attribute to a property of type `Dictionary<string,object>` or `Dictionary<string,JsonElement>`:

C#

```
public class WeatherForecastWithExtensionData
{
    public DateTimeOffset Date { get; set; }
    public int TemperatureCelsius { get; set; }
    public string? Summary { get; set; }
    [JsonExtensionData]
    public Dictionary<string, JsonElement>? ExtensionData { get; set; }
}
```

The following table shows the result of deserializing the JSON shown earlier into this sample type. The extra data becomes key-value pairs of the `ExtensionData` property:

[+] Expand table

| Property | Value | Notes |
|--------------------|--|---|
| Date | "8/1/2019 12:00:00 AM -07:00" | |
| TemperatureCelsius | 0 | Case-sensitive mismatch (<code>temperatureCelsius</code> in the JSON), so the property isn't set. |
| Summary | "Hot" | |
| ExtensionData | {"temperatureCelsius": 25, "DatesAvailable": ["2019-08-01T00:00:00-07:00", "2019-08-02T00:00:00-07:00"], "SummaryWords": ["Cool", "Windy", "Humid"]} | Since the case didn't match, <code>temperatureCelsius</code> is an extra and becomes a key-value pair in the dictionary. Each extra array from the JSON becomes a key-value pair, with an array as the value object. |

When the target object is serialized, the extension data key value pairs become JSON properties just as they were in the incoming JSON:

JSON

```
{
    "Date": "2019-08-01T00:00:00-07:00",
    "TemperatureCelsius": 0,
    "Summary": "Hot",
```

```

    "temperatureCelsius": 25,
    "DatesAvailable": [
        "2019-08-01T00:00:00-07:00",
        "2019-08-02T00:00:00-07:00"
    ],
    "SummaryWords": [
        "Cool",
        "Windy",
        "Humid"
    ]
}

```

Notice that the `ExtensionData` property name doesn't appear in the JSON. This behavior lets the JSON make a round trip without losing any extra data that otherwise wouldn't be serialized.

The following example shows a round trip from JSON to a deserialized object and back to JSON:

C#

```

using System.Text.Json;
using System.Text.Json.Serialization;

namespace RoundtripExtensionData
{
    public class WeatherForecast
    {
        public DateTimeOffset Date { get; set; }
        public int TemperatureCelsius { get; set; }
        public string? Summary { get; set; }
        [JsonExtensionData]
        public Dictionary<string, JsonElement>? ExtensionData { get; set; }
    }
    public class Program
    {
        public static void Main()
        {
            string jsonString =
@"
{
    ""Date"": ""2019-08-01T00:00:00-07:00"",
    ""temperatureCelsius"": 25,
    ""Summary"": """",
    ""SummaryField"": """",
    ""DatesAvailable"": [
        ""2019-08-01T00:00:00-07:00"",
        ""2019-08-02T00:00:00-07:00"""
    ],
    ""SummaryWords"": [
        """Cool""",
        """Windy""",
        """Humid"""
    ]
}

```

```
        ]
    }";
    WeatherForecast weatherForecast =
        JsonSerializer.Deserialize<WeatherForecast>(jsonString)!;

    var serializeOptions = new JsonSerializerOptions { WriteIndented
= true };
    jsonString = JsonSerializer.Serialize(weatherForecast,
serializeOptions);
    Console.WriteLine($"JSON output:\n{jsonString}\n");
}
}

// output:
//JSON output:
//{
//  "Date": "2019-08-01T00:00:00-07:00",
//  "TemperatureCelsius": 0,
//  "Summary": "Hot",
//  "temperatureCelsius": 25,
//  "SummaryField": "Hot",
//  "DatesAvailable": [
//    "2019-08-01T00:00:00-07:00",
//    "2019-08-02T00:00:00-07:00"
//  ],
//  "SummaryWords": [
//    "Cool",
//    "Windy",
//    "Humid"
//  ]
//}
```

Deserialize into JsonElement or JsonNode

If you just want to be flexible about what JSON is acceptable for a particular property, an alternative is to deserialize into [JsonElement](#) or [JsonNode](#). Any valid JSON property can be deserialized into [JsonElement](#) or [JsonNode](#). Choose [JsonElement](#) to create an immutable object or [JsonNode](#) to create a mutable object.

The following example shows a round trip from JSON and back to JSON for a class that includes properties of type [JsonElement](#) and [JsonNode](#).

C#

```
using System.Text.Json;
using System.Text.Json.Nodes;

namespace RoundtripJsonElementAndNode
{
    public class WeatherForecast
```

```

    {
        public DateTimeOffset Date { get; set; }
        public int TemperatureCelsius { get; set; }
        public string? Summary { get; set; }
        public JsonElement DatesAvailable { get; set; }
        public JsonNode? SummaryWords { get; set; }
    }
    public class Program
    {
        public static void Main()
        {
            string jsonString =
@"{  

    ""Date"": ""2019-08-01T00:00:00-07:00"",
    ""TemperatureCelsius"": 25,
    ""Summary"": ""Hot"",
    ""DatesAvailable"": [
        ""2019-08-01T00:00:00-07:00"",
        ""2019-08-02T00:00:00-07:00""
    ],
    ""SummaryWords"": [
        ""Cool"",
        ""Windy"",
        ""Humid""
    ]
}";
            WeatherForecast? weatherForecast =
                JsonSerializer.Deserialize<WeatherForecast>(jsonString);

            var serializeOptions = new JsonSerializerOptions { WriteIndented
= true };
            jsonString = JsonSerializer.Serialize(weatherForecast,
serializeOptions);
            Console.WriteLine(jsonString);
        }
    }
// output:
//{
//    "Date": "2019-08-01T00:00:00-07:00",
//    "TemperatureCelsius": 25,
//    "Summary": "Hot",
//    "DatesAvailable": [
//        "2019-08-01T00:00:00-07:00",
//        "2019-08-02T00:00:00-07:00"
//    ],
//    "SummaryWords": [
//        "Cool",
//        "Windy",
//        "Humid"
//    ]
//}

```

See also

- [System.Text.Json overview](#)
- [How to serialize and deserialize JSON](#)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

.NET is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

Use immutable types and properties

Article • 10/25/2023

An immutable *type* is one that prevents you from changing any property or field values of an object after it's instantiated. The type might be a record, have no public properties or fields, have read-only properties, or have properties with private or init-only setters. [System.String](#) is an example of an immutable type. [System.Text.Json](#) provides different ways that you can deserialize JSON to immutable types.

Parameterized constructors

By default, [System.Text.Json](#) uses the default public parameterless constructor.

However, you can tell it to use a parameterized constructor, which makes it possible to deserialize an immutable class or struct.

- For a class, if the only constructor is a parameterized one, that constructor will be used.
- For a struct, or a class with multiple constructors, specify the one to use by applying the [\[JsonConstructor\]](#) attribute. When the attribute is not used, a public parameterless constructor is always used if present.

The following example uses the [\[JsonConstructor\]](#) attribute:

C#

```
using System.Text.Json;
using System.Text.Json.Serialization;

namespace ImmutableTypes
{
    public struct Forecast
    {
        public DateTime Date { get; }
        public int TemperatureC { get; }
        public string Summary { get; }

        [JsonConstructor]
        public Forecast(DateTime date, int temperatureC, string
summary) =>
            (Date, TemperatureC, Summary) = (date, temperatureC,
summary);
    }

    public class Program
    {
```

```

        public static void Main()
    {
        var json = @"{""date"":""2020-09-06T11:31:01.923395-
07:00"",""temperatureC"":-1,""summary"":""Cold""} ";
        Console.WriteLine($"Input JSON: {json}");

        var options = new
JsonSerializerOptions(JsonSerializerDefaults.Web);

        var forecast = JsonSerializer.Deserialize<Forecast>(json,
options);

        Console.WriteLine($"forecast.Date: {forecast.Date}");
        Console.WriteLine($"forecast.TemperatureC:
{forecast.TemperatureC}");
        Console.WriteLine($"forecast.Summary: {forecast.Summary}");

        var roundTrippedJson =
JsonSerializer.Serialize<Forecast>(forecast, options);

        Console.WriteLine($"Output JSON: {roundTrippedJson}");

    }
}

// Produces output like the following example:
//
//Input JSON: { "date":"2020-09-06T11:31:01.923395-
07:00","temperatureC":-1,"summary":"Cold"}
//forecast.Date: 9 / 6 / 2020 11:31:01 AM
//forecast.TemperatureC: -1
//forecast.Summary: Cold
//Output JSON: { "date":"2020-09-06T11:31:01.923395-
07:00","temperatureC":-1,"summary":"Cold"}
```

In .NET 7 and earlier versions, the `[JsonConstructor]` attribute can only be used with public constructors.

The parameter names of a parameterized constructor must match the property names and types. Matching is case-insensitive, and the constructor parameter must match the actual property name even if you use `[JsonPropertyName]` to rename a property. In the following example, the name for the `TemperatureC` property is changed to `celsius` in the JSON, but the constructor parameter is still named `temperaturec`:

C#

```

using System.Text.Json;
using System.Text.Json.Serialization;

namespace ImmutableTypesCtorParams
```

```

{
    public struct Forecast
    {
        public DateTime Date { get; }
        [JsonPropertyName("celsius")]
        public int TemperatureC { get; }
        public string Summary { get; }

        [JsonConstructor]
        public Forecast(DateTime date, int temperatureC, string summary) =>
            (Date, TemperatureC, Summary) = (date, temperatureC, summary);
    }
}

public class Program
{
    public static void Main()
    {
        var json = @"{""date"":""2020-09-06T11:31:01.923395-
07:00"",""celsius"":-1,""summary"":""Cold""} ";
        Console.WriteLine($"Input JSON: {json}");

        var options = new
JsonSerializerOptions(JsonSerializerDefaults.Web);

        var forecast = JsonSerializer.Deserialize<Forecast>(json,
options);

        Console.WriteLine($"forecast.Date: {forecast.Date}");
        Console.WriteLine($"forecast.TemperatureC:
{forecast.TemperatureC}");
        Console.WriteLine($"forecast.Summary: {forecast.Summary}");

        var roundTrippedJson =
            JsonSerializer.Serialize<Forecast>(forecast, options);

        Console.WriteLine($"Output JSON: {roundTrippedJson}");

    }
}
}

// Produces output like the following example:
//
//Input JSON: { "date":"2020-09-06T11:31:01.923395-
07:00","celsius":-1,"summary":"Cold"}
//forecast.Date: 9 / 6 / 2020 11:31:01 AM
//forecast.TemperatureC: -1
//forecast.Summary: Cold
//Output JSON: { "date":"2020-09-06T11:31:01.923395-
07:00","celsius":-1,"summary":"Cold"}
```

Besides `[JsonPropertyName]`, the following attributes support deserialization with parameterized constructors:

- [\[JsonConverter\]](#)
- [\[JsonIgnore\]](#)
- [\[JsonInclude\]](#)
- [\[JsonNumberHandling\]](#)

Records

Records are also supported for both serialization and deserialization, as shown in the following example:

C#

```
using System.Text.Json;

namespace Records
{
    public record Forecast(DateTime Date, int TemperatureC)
    {
        public string? Summary { get; init; }
    };

    public class Program
    {
        public static void Main()
        {
            Forecast forecast = new(DateTime.Now, 40)
            {
                Summary = "Hot!"
            };

            string forecastJson = JsonSerializer.Serialize<Forecast>
(forecast);
            Console.WriteLine(forecastJson);
            Forecast? forecastObj = JsonSerializer.Deserialize<Forecast>
(forecastJson);
            Console.WriteLine(forecastObj);
        }
    }
}

// Produces output like the following example:
// 
//{"Date":"2020-10-21T15:26:10.5044594-
07:00","TemperatureC":40,"Summary":"Hot!"}
//Forecast { Date = 10 / 21 / 2020 3:26:10 PM, TemperatureC = 40, Summary =
Hot! }
```

You can apply any of the attributes to the property names, using the `property:` target on the attribute. For more information on positional records, see the article on [records](#)

in the C# language reference.

Non-public members and property accessors

You can enable use of a non-public *accessor* on a property by using the [\[JsonInclude\]](#) attribute, as shown in the following example:

C#

```
using System.Text.Json;
using System.Text.Json.Serialization;

namespace NonPublicAccessors
{
    public class Forecast
    {
        public DateTime Date { get; init; }

        [JsonInclude]
        public int TemperatureC { get; private set; }

        [JsonInclude]
        public string? Summary { private get; set; }
    };

    public class Program
    {
        public static void Main()
        {
            string json = @"{""Date"":""2020-10-23T09:51:03.8702889-07:00"",""TemperatureC"":40,""Summary"":""Hot""}";
            Console.WriteLine($"Input JSON: {json}");

            Forecast forecastDeserialized =
JsonSerializer.Deserialize<Forecast>(json)!;
            Console.WriteLine($"Date: {forecastDeserialized.Date}");
            Console.WriteLine($"TemperatureC: {forecastDeserialized.TemperatureC}");

            json = JsonSerializer.Serialize<Forecast>(forecastDeserialized);
            Console.WriteLine($"Output JSON: {json}");
        }
    }
}

// Produces output like the following example:
//
//Input JSON: { "Date":"2020-10-23T09:51:03.8702889-07:00","TemperatureC":40,"Summary":"Hot"}
//Date: 10 / 23 / 2020 9:51:03 AM
//TemperatureC: 40
```

```
//Output JSON: { "Date":"2020-10-23T09:51:03.8702889-07:00", "TemperatureC":40, "Summary":"Hot"}
```

By including a property with a private setter, you can still deserialize that property.

In .NET 8 and later versions, you can also use the [\[JsonInclude\]](#) attribute to opt non-public *members* into the serialization contract for a given type.

ⓘ Note

In source-generation mode, you can't serialize `private` members or use `private` accessors by annotating them with the [\[JsonInclude\]](#) attribute. And you can only serialize `internal` members or use `internal` accessors if they're in the same assembly as the generated [JsonSerializerContext](#).

Read-only properties

In .NET 8 and later versions, read-only properties, or those that have no setter either private or public, can also be deserialized. While you can't change the instance that the property references, if the type of the property is mutable, you can modify it. For example, you can add an element to a list. To deserialize a read-only property, you need to set its object creation handling behavior to *populate* instead of *replace*. For example, you can annotate the property with the [JsonObjectCreationHandlingAttribute](#) attribute.

C#

```
class A
{
    [JsonObjectCreationHandling(JsonObjectCreationHandling.Populate)]
    public List<int> Numbers1 { get; } = new List<int>() { 1, 2, 3 };
}
```

For more information, see [Populate initialized properties](#).

See also

- [System.Text.Json overview](#)
- [How to serialize and deserialize JSON](#)

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

Populate initialized properties

Article • 10/25/2023

Starting in .NET 8, you can specify a preference to either [replace](#) or [populate](#) .NET properties when JSON is deserialized. The [JsonObjectCreationHandling](#) enum provides the object creation handling choices:

- [JsonObjectCreationHandling.Replace](#)
- [JsonObjectCreationHandling.Populate](#)

Default (replace) behavior

The `System.Text.Json` deserializer always creates a new instance of the target type. However, even though a new instance is created, some properties and fields might already be initialized as part of the object's construction. Consider the following type:

C#

```
class A
{
    public List<int> Numbers1 { get; } = [1, 2, 3];
    public List<int> Numbers2 { get; set; } = [1, 2, 3];
}
```

When you create an instance of this class, the `Numbers1` (and `Numbers2`) property's value is a list with three elements (1, 2, and 3). If you deserialize JSON to this type, the *default* behavior is that property values are *replaced*:

- For `Numbers1`, since it's read-only (no setter), it still has the values 1, 2, and 3 in its list.
- For `Numbers2`, which is read-write, a new list is allocated and the values from the JSON are added.

For example, if you execute the following deserialization code, `Numbers1` contains the values 1, 2, and 3 and `Numbers2` contains the values 4, 5, and 6.

C#

```
A? a = JsonSerializer.Deserialize<A>("""" {"Numbers1": [4,5,6], "Numbers2": [4,5,6]} """");
```

Populate behavior

Starting in .NET 8, you can change the deserialization behavior to modify (*populate*) properties and fields instead of replace them:

- For a collection type property, the object is reused without clearing. If the collection is prepopulated with elements, they'll show in the final deserialized result along with the values from the JSON. For an example, see [Collection property example](#).
- For a property that's an object with properties, its mutable properties are updated to the JSON values but the object reference itself doesn't change.
- For a struct type property, the effective behavior is that for its mutable properties, any existing values are kept and new values from the JSON are added. However, unlike a reference property, the object itself isn't reused since it's a value type. Instead, a copy of the struct is modified and then reassigned to the property. For an example, see [Struct property example](#).

A struct property must have a setter; otherwise, an [InvalidOperationException](#) is thrown at run time.

ⓘ Note

The populate behavior currently doesn't work for types that have a parameterized constructor. For more information, see [dotnet/runtime issue 92877](#).

Read-only properties

For populating reference properties that are mutable, since the instance that the property references isn't *replaced*, the property doesn't need to have a setter. This behavior means that deserialization can also populate *read-only* properties.

ⓘ Note

Struct properties still require setters because the instance is replaced with a modified copy.

Collection property example

Consider the same class `A` from the [replace behavior](#) example, but this time annotated with a preference for *populating* properties instead of replacing them:

```
C#  
  
[JsonObjectCreationHandling(JsonObjectCreationHandling.Populate)]  
class A  
{  
    public List<int> Numbers1 { get; } = [1, 2, 3];  
    public List<int> Numbers2 { get; set; } = [1, 2, 3];  
}
```

If you execute the following deserialization code, both `Numbers1` and `Numbers2` contain the values 1, 2, 3, 4, 5, and 6:

```
C#  
  
A? a = JsonSerializer.Deserialize<A>( """{ "Numbers1": [4,5,6], "Numbers2": [4,5,6] } """ );
```

Struct property example

The following class contains a struct property, `s1`, whose deserialization behavior is set to [Populate](#). After executing this code, `c.s1.Value1` has a value of 10 (from the constructor) and `c.s1.Value2` has a value of 5 (from the JSON).

```
C#  
  
C? c = JsonSerializer.Deserialize<C>( """{ "S1": { "Value2": 5 } } """ );  
  
class C  
{  
    public C()  
    {  
        _s1 = new S  
        {  
            Value1 = 10  
        };  
    }  
  
    private S _s1;  
  
    [JsonObjectCreationHandling(JsonObjectCreationHandling.Populate)]  
    public S S1  
    {  
        get { return _s1; }  
        set { _s1 = value; }  
    }
```

```
}
```



```
struct S
```

```
{
```

```
    public int Value1 { get; set; }
```

```
    public int Value2 { get; set; }
```

```
}
```

If the default [Replace](#) behavior was used instead, `c.S1.Value1` would have its default value of 0 after deserialization. That's because the constructor `c()` would be called, setting `c.S1.Value1` to 10, but then the value of S1 would be replaced with a new instance. (`c.S1.Value2` would still be 5, since the JSON replaces the default value.)

How to specify

There are multiple ways to specify a preference for *replace* or *populate*:

- Use the [JsonObjectCreationHandlingAttribute](#) attribute to annotate at the type or property level. If you set the attribute at the type level and set its [Handling](#) property to [Populate](#), the behavior will only apply to those properties where population is possible (for example, value types must have a setter).

If you want the type-wide preference to be [Populate](#), but want to exclude one or more properties from that behavior, you can add the attribute at the type level and again at the property level to override the inherited behavior. That pattern is shown in the following code.

C#

```
// Type-level preference is Populate.  
[JsonObjectCreationHandling(JsonObjectCreationHandling.Populate)]  
class B  
{  
    // For this property only, use Replace behavior.  
    [JsonObjectCreationHandling(JsonObjectCreationHandling.Replace)]  
    public List<int> Numbers1 { get; } = [1, 2, 3];  
    public List<int> Numbers2 { get; set; } = [1, 2, 3];  
}
```

- Set [JsonSerializerOptions.PreferredObjectCreationHandling](#) (or, for source generation, [JsonSourceGenerationOptionsAttribute.PreferredObjectCreationHandling](#)) to specify a global preference.

C#

```
var options = new JsonSerializerOptions
{
    PreferredObjectCreationHandling =
    JsonObjectCreationHandling.Populate
};
```

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

Migrate from Newtonsoft.Json to System.Text.Json

Article • 10/20/2023

This article shows how to migrate from [Newtonsoft.Json](#) to [System.Text.Json](#).

The `System.Text.Json` namespace provides functionality for serializing to and deserializing from JavaScript Object Notation (JSON). The `System.Text.Json` library is included in the runtime for [.NET Core 3.1](#) and later versions. For other target frameworks, install the [System.Text.Json](#) NuGet package. The package supports:

- .NET Standard 2.0 and later versions
- .NET Framework 4.7.2 and later versions
- .NET Core 2.0, 2.1, and 2.2

`System.Text.Json` focuses primarily on performance, security, and standards compliance. It has some key differences in default behavior and doesn't aim to have feature parity with `Newtonsoft.Json`. For some scenarios, `System.Text.Json` currently has no built-in functionality, but there are recommended workarounds. For other scenarios, workarounds are impractical.

We're investing in adding the features that are most often requested. If your application depends on a missing feature, consider [filing an issue](#) in the dotnet/runtime GitHub repository to find out if support for your scenario can be added.

Most of this article is about how to use the `JsonSerializer` API, but it also includes guidance on how to use the `JsonDocument` (which represents the Document Object Model or DOM), `Utf8JsonReader`, and `Utf8JsonWriter` types.

In Visual Basic, you can't use `Utf8JsonReader`, which also means you can't write custom converters. Most of the workarounds presented here require that you write custom converters. You can write a custom converter in C# and register it in a Visual Basic project. For more information, see [Visual Basic support](#).

Table of differences

The following table lists `Newtonsoft.Json` features and `System.Text.Json` equivalents.

The equivalents fall into the following categories:

- Supported by built-in functionality. Getting similar behavior from `System.Text.Json` may require the use of an attribute or global option.

- ⚠ Not supported, but workaround is possible. The workarounds are [custom converters](#), which may not provide complete parity with `Newtonsoft.Json` functionality. For some of these, sample code is provided as examples. If you rely on these `Newtonsoft.Json` features, migration will require modifications to your .NET object models or other code changes.
- ✗ Not supported, and workaround is not practical or possible. If you rely on these `Newtonsoft.Json` features, migration will not be possible without significant changes.

| Newtonsoft.Json feature | System.Text.Json equivalent |
|--|---|
| Case-insensitive deserialization by default | ✓ PropertyNameCaseInsensitive global setting |
| Camel-case property names | ✓ PropertyNamingPolicy global setting |
| Snake-case property names | ✓ Snake case naming policy |
| Minimal character escaping | ✓ Strict character escaping, configurable |
| <code>NullValueHandling.Ignore</code> global setting | ✓ DefaultIgnoreCondition global option |
| Allow comments | ✓ ReadCommentHandling global setting |
| Allow trailing commas | ✓ AllowTrailingCommas global setting |
| Custom converter registration | ✓ Order of precedence differs |
| No maximum depth by default | ✓ Default maximum depth 64, configurable |
| <code>PreserveReferencesHandling</code> global setting | ✓ ReferenceHandling global setting |
| Serialize or deserialize numbers in quotes | ✓ NumberHandling global setting, [JsonNumberHandling] attribute |
| Deserialize to immutable classes and structs | ✓ JsonConstructor , C# 9 Records |
| Support for fields | ✓ IncludeFields global setting, [JsonInclude] attribute |
| <code>DefaultValueHandling</code> global setting | ✓ DefaultIgnoreCondition global setting |
| <code>NullValueHandling</code> setting on <code>[JsonProperty]</code> | ✓ JsonIgnore attribute |
| <code>DefaultValueHandling</code> setting on <code>[JsonProperty]</code> | ✓ JsonIgnore attribute |
| Deserialize <code>Dictionary</code> with non-string key | ✓ Supported |

| Newtonsoft.Json feature | System.Text.Json equivalent |
|--|---|
| Support for non-public property setters and getters | ✓ JsonInclude attribute |
| <code>[JsonConstructor]</code> attribute | ✓ [JsonConstructor] attribute |
| <code>ReferenceLoopHandling</code> global setting | ✓ ReferenceHandling global setting |
| Callbacks | ✓ Callbacks |
| NaN, Infinity, -Infinity | ✓ Supported |
| <code>Required</code> setting on <code>[JsonProperty]</code> attribute | ✓ [JsonRequired] attribute and C# required modifier |
| <code>DefaultContractResolver</code> to ignore properties | ✓ DefaultJsonTypeInfoResolver class |
| Polymorphic serialization | ✓ [JsonDerivedType] attribute |
| Polymorphic deserialization | ✓ Type discriminator on [JsonDerivedType] attribute |
| Deserialize string enum value | ✓ Deserialize string enum values |
| <code>MissingMemberHandling</code> global setting | ✓ Handle missing members |
| Populate properties without setters | ✓ Populate properties without setters |
| <code>ObjectCreationHandling</code> global setting | ✓ Reuse rather than replace properties |
| Support for a broad range of types | ⚠ Some types require custom converters |
| Deserialize inferred type to <code>object</code> properties | ⚠ Not supported, workaround, sample |
| Deserialize JSON <code>null</code> literal to non-nullable value types | ⚠ Not supported, workaround, sample |
| <code>DateTimeZoneHandling</code> , <code>DateFormatString</code> settings | ⚠ Not supported, workaround, sample |
| <code>JsonConvert.PopulateObject</code> method | ⚠ Not supported, workaround |
| Support for <code>System.Runtime.Serialization</code> attributes | ⚠ Not supported, workaround, sample |
| <code>JsonObjectAttribute</code> | ⚠ Not supported, workaround |
| Allow property names without quotes | ✗ Not supported by design |

| Newtonsoft.Json feature | System.Text.Json equivalent |
|--|-----------------------------|
| Allow single quotes around string values | ✖ Not supported by design |
| Allow non-string JSON values for string properties | ✖ Not supported by design |
| TypeNameHandling.All global setting | ✖ Not supported by design |
| Support for <code>JsonPath</code> queries | ✖ Not supported |
| Configurable limits | ✖ Not supported |

This is not an exhaustive list of `Newtonsoft.Json` features. The list includes many of the scenarios that have been requested in [GitHub issues](#) or [StackOverflow](#) posts. If you implement a workaround for one of the scenarios listed here that doesn't currently have sample code, and if you want to share your solution, select [This page](#) in the [Feedback](#) section at the bottom of this page. That creates an issue in this documentation's GitHub repo and lists it in the [Feedback](#) section on this page too.

Differences in default behavior

`System.Text.Json` is strict by default and avoids any guessing or interpretation on the caller's behalf, emphasizing deterministic behavior. The library is intentionally designed this way for performance and security. `Newtonsoft.Json` is flexible by default. This fundamental difference in design is behind many of the following specific differences in default behavior.

Case-insensitive deserialization

During deserialization, `Newtonsoft.Json` does case-insensitive property name matching by default. The `System.Text.Json` default is case-sensitive, which gives better performance since it's doing an exact match. For information about how to do case-insensitive matching, see [Case-insensitive property matching](#).

If you're using `System.Text.Json` indirectly by using ASP.NET Core, you don't need to do anything to get behavior like `Newtonsoft.Json`. ASP.NET Core specifies the settings for camel-casing property names and case-insensitive matching when it uses `System.Text.Json`.

ASP.NET Core also enables deserializing [quoted numbers](#) by default.

Minimal character escaping

During serialization, `Newtonsoft.Json` is relatively permissive about letting characters through without escaping them. That is, it doesn't replace them with `\uxxxx` where `xxxx` is the character's code point. Where it does escape them, it does so by emitting a `\` before the character (for example, `"` becomes `\"`). `System.Text.Json` escapes more characters by default to provide defense-in-depth protections against cross-site scripting (XSS) or information-disclosure attacks and does so by using the six-character sequence. `System.Text.Json` escapes all non-ASCII characters by default, so you don't need to do anything if you're using `StringEscapeHandling.EscapeNonAscii` in `Newtonsoft.Json`. `System.Text.Json` also escapes HTML-sensitive characters, by default. For information about how to override the default `System.Text.Json` behavior, see [Customize character encoding](#).

Comments

During deserialization, `Newtonsoft.Json` ignores comments in the JSON by default. The `System.Text.Json` default is to throw exceptions for comments because the [RFC 8259](#) specification doesn't include them. For information about how to allow comments, see [Allow comments and trailing commas](#).

Trailing commas

During deserialization, `Newtonsoft.Json` ignores trailing commas by default. It also ignores multiple trailing commas (for example, `[{"Color":"Red"}, {"Color":"Green"}, ,]`). The `System.Text.Json` default is to throw exceptions for trailing commas because the [RFC 8259](#) specification doesn't allow them. For information about how to make `System.Text.Json` accept them, see [Allow comments and trailing commas](#). There's no way to allow multiple trailing commas.

Converter registration precedence

The `Newtonsoft.Json` registration precedence for custom converters is as follows:

- Attribute on property
- Attribute on type
- [Converters](#) collection

This order means that a custom converter in the `Converters` collection is overridden by a converter that is registered by applying an attribute at the type level. Both of those registrations are overridden by an attribute at the property level.

The `System.Text.Json` registration precedence for custom converters is different:

- Attribute on property
- `Converters` collection
- Attribute on type

The difference here is that a custom converter in the `Converters` collection overrides an attribute at the type level. The intention behind this order of precedence is to make runtime changes override design-time choices. There's no way to change the precedence.

For more information about custom converter registration, see [Register a custom converter](#).

Maximum depth

The latest version of `Newtonsoft.Json` has a maximum depth limit of 64 by default.

`System.Text.Json` also has a default limit of 64, and it's configurable by setting `JsonSerializerOptions.MaxDepth`.

If you're using `System.Text.Json` indirectly by using ASP.NET Core, the default maximum depth limit is 32. The default value is the same as for model binding and is set in the `JsonOptions class` ↗.

JSON strings (property names and string values)

During deserialization, `Newtonsoft.Json` accepts property names surrounded by double quotes, single quotes, or without quotes. It accepts string values surrounded by double quotes or single quotes. For example, `Newtonsoft.Json` accepts the following JSON:

```
JSON
{
  "name1": "value",
  'name2': "value",
  name3: 'value'
}
```

`System.Text.Json` only accepts property names and string values in double quotes because that format is required by the [RFC 8259](#) ↗ specification and is the only format considered valid JSON.

A value enclosed in single quotes results in a `JsonException` with the following message:

Output

```
''' is an invalid start of a value.
```

Non-string values for string properties

`Newtonsoft.Json` accepts non-string values, such as a number or the literals `true` and `false`, for deserialization to properties of type `string`. Here's an example of JSON that `Newtonsoft.Json` successfully deserializes to the following class:

JSON

```
{
    "String1": 1,
    "String2": true,
    "String3": false
}
```

C#

```
public class ExampleClass
{
    public string String1 { get; set; }
    public string String2 { get; set; }
    public string String3 { get; set; }
}
```

`System.Text.Json` doesn't deserialize non-string values into `string` properties. A non-string value received for a `string` field results in a `JsonException` with the following message:

Output

```
The JSON value could not be converted to System.String.
```

Scenarios using `JsonSerializer`

Some of the following scenarios aren't supported by built-in functionality, but workarounds are possible. The workarounds are [custom converters](#), which may not provide complete parity with `Newtonsoft.Json` functionality. For some of these, sample code is provided as examples. If you rely on these `Newtonsoft.Json` features, migration will require modifications to your .NET object models or other code changes.

For some of the following scenarios, workarounds are not practical or possible. If you rely on these `Newtonsoft.Json` features, migration will not be possible without significant changes.

Allow or write numbers in quotes

`Newtonsoft.Json` can serialize or deserialize numbers represented by JSON strings (surrounded by quotes). For example, it can accept: `{"DegreesCelsius": "23"}` instead of `{"DegreesCelsius": 23}`. To enable that behavior in `System.Text.Json`, set `JsonSerializerOptions.NumberHandling` to `WriteAsString` or `AllowReadingFromString`, or use the `[JsonNumberHandling]` attribute.

If you're using `System.Text.Json` indirectly by using ASP.NET Core, you don't need to do anything to get behavior like `Newtonsoft.Json`. ASP.NET Core specifies [web defaults](#) when it uses `System.Text.Json`, and web defaults allow quoted numbers.

For more information, see [Allow or write numbers in quotes](#).

Specify constructor to use when deserializing

The `Newtonsoft.Json [JsonConstructor]` attribute lets you specify which constructor to call when deserializing to a POCO.

`System.Text.Json` also has a `[JsonConstructor]` attribute. For more information, see [Immutable types and Records](#).

Conditionally ignore a property

`Newtonsoft.Json` has several ways to conditionally ignore a property on serialization or deserialization:

- `DefaultContractResolver` lets you select properties to include or ignore, based on arbitrary criteria.
- The `NullValueHandling` and `DefaultValueHandling` settings on `JsonSerializerSettings` let you specify that all null-value or default-value properties should be ignored.
- The `NullValueHandling` and `DefaultValueHandling` settings on the `[JsonProperty]` attribute let you specify individual properties that should be ignored when set to null or the default value.

`System.Text.Json` provides the following ways to ignore properties or fields while serializing:

- The `[JsonIgnore]` attribute on a property causes the property to be omitted from the JSON during serialization.
- The `IgnoreReadOnlyProperties` global option lets you ignore all read-only properties.
- If you're [including fields](#), the `JsonSerializerOptions.IgnoreReadOnlyFields` global option lets you ignore all read-only fields.
- The `DefaultIgnoreCondition` global option lets you [ignore all value type properties that have default values](#), or [ignore all reference type properties that have null values](#).

In addition, in .NET 7 and later versions, you can customize the JSON contract to ignore properties based on arbitrary criteria. For more information, see [Custom contracts](#).

Public and non-public fields

`Newtonsoft.Json` can serialize and deserialize fields as well as properties.

In `System.Text.Json`, use the `JsonSerializerOptions.IncludeFields` global setting or the `[JsonInclude]` attribute to include public fields when serializing or deserializing. For an example, see [Include fields](#).

Preserve object references and handle loops

By default, `Newtonsoft.Json` serializes by value. For example, if an object contains two properties that contain a reference to the same `Person` object, the values of that `Person` object's properties are duplicated in the JSON.

`Newtonsoft.Json` has a `PreserveReferencesHandling` setting on `JsonSerializerSettings` that lets you serialize by reference:

- An identifier metadata is added to the JSON created for the first `Person` object.
- The JSON that is created for the second `Person` object contains a reference to that identifier instead of property values.

`Newtonsoft.Json` also has a `ReferenceLoopHandling` setting that lets you ignore circular references rather than throw an exception.

To preserve references and handle circular references in `System.Text.Json`, set `JsonSerializerOptions.ReferenceHandler` to `Preserve`. The `ReferenceHandler.Preserve`

setting is equivalent to `PreserveReferencesHandling = PreserveReferencesHandling.All` in `Newtonsoft.Json`.

The `ReferenceHandler.IgnoreCycles` option has behavior similar to `Newtonsoft.Json ReferenceLoopHandling.Ignore`. One difference is that the `System.Text.Json` implementation replaces reference loops with the `null` JSON token instead of ignoring the object reference. For more information, see [Ignore circular references](#).

Like the `Newtonsoft.Json ReferenceResolver`, the `System.Text.Json.Serialization.ReferenceResolver` class defines the behavior of preserving references on serialization and deserialization. Create a derived class to specify custom behavior. For an example, see [GuidReferenceResolver](#).

Some related `Newtonsoft.Json` features aren't supported:

- [JsonPropertyAttribute.IsReference](#)
- [JsonPropertyAttribute.ReferenceLoopHandling](#)

For more information, see [Preserve references and handle circular references](#).

Dictionary with non-string key

Both `Newtonsoft.Json` and `System.Text.Json` support collections of type `Dictionary<TKey, TValue>`. However, in `System.Text.Json`, `TKey` must be a primitive type, not a custom type. For more information, see [Supported key types](#).

⊗ Caution

Deserializing to a `Dictionary<TKey, TValue>` where `TKey` is typed as anything other than `string` could introduce a security vulnerability in the consuming application. For more information, see [dotnet/runtime#4761](#).

Types without built-in support

`System.Text.Json` doesn't provide built-in support for the following types:

- `DataTable` and related types (for more information, see [Supported collection types](#))
- `ExpandoObject`
- `TimeZoneInfo`
- `BigInteger`
- `DBNull`

- [Type](#)
- [ValueTuple](#) and its associated generic types

Custom converters can be implemented for types that don't have built-in support.

Polymorphic serialization

[Newtonsoft.Json](#) automatically does polymorphic serialization. Starting in .NET 7, [System.Text.Json](#) supports polymorphic serialization through the [JsonDerivedTypeAttribute](#) attribute. For more information, see [Serialize properties of derived classes](#).

Polymorphic deserialization

[Newtonsoft.Json](#) has a [TypeNameHandling](#) setting that adds type-name metadata to the JSON while serializing. It uses the metadata while deserializing to do polymorphic deserialization. Starting in .NET 7, [System.Text.Json](#) relies on type discriminator information to perform polymorphic deserialization. This metadata is emitted in the JSON and then used during deserialization to determine whether to deserialize to the base type or a derived type. For more information, see [Serialize properties of derived classes](#).

To support polymorphic deserialization in older .NET versions, create a converter like the example in [How to write custom converters](#).

Deserialize string enum values

By default, [System.Text.Json](#) doesn't support deserializing string enum values, whereas [Newtonsoft.Json](#) does. For example, the following code throws a [JsonException](#):

C#

```
string json = "{ \"Text\": \"Hello\", \"Enum\": \"Two\" }";
var _ = JsonSerializer.Deserialize<MyObj>(json); // Throws exception.

class MyObj
{
    public string Text { get; set; } = "";
    public MyEnum Enum { get; set; }
}

enum MyEnum
{
    One,
    Two,
```

```
    Three  
}
```

However, you can enable deserialization of string enum values by using the [JsonStringEnumConverter](#) converter. For more information, see [Enums as strings](#).

Deserialization of object properties

When `Newtonsoft.Json` deserializes to [Object](#), it:

- Infers the type of primitive values in the JSON payload (other than `null`) and returns the stored `string`, `long`, `double`, `boolean`, or `DateTime` as a boxed object. *Primitive values* are single JSON values such as a JSON number, string, `true`, `false`, or `null`.
- Returns a `JObject` or `JArray` for complex values in the JSON payload. *Complex values* are collections of JSON key-value pairs within braces (`{}`) or lists of values within brackets (`[]`). The properties and values within the braces or brackets can have additional properties or values.
- Returns a null reference when the payload has the `null` JSON literal.

`System.Text.Json` stores a boxed `JsonElement` for both primitive and complex values whenever deserializing to [Object](#), for example:

- An `object` property.
- An `object` dictionary value.
- An `object` array value.
- A root `object`.

However, `System.Text.Json` treats `null` the same as `Newtonsoft.Json` and returns a null reference when the payload has the `null` JSON literal in it.

To implement type inference for `object` properties, create a converter like the example in [How to write custom converters](#).

Deserialize null to non-nullable type

`Newtonsoft.Json` doesn't throw an exception in the following scenario:

- `NullValueHandling` is set to `Ignore`, and
- During deserialization, the JSON contains a null value for a non-nullable value type.

In the same scenario, `System.Text.Json` does throw an exception. (The corresponding null-handling setting in `System.Text.Json` is `JsonSerializerOptions.IgnoreNullValues = true`.)

If you own the target type, the best workaround is to make the property in question nullable (for example, change `int` to `int?`).

Another workaround is to make a converter for the type, such as the following example that handles null values for `DateTimeOffset` types:

C#

```
using System.Text.Json;
using System.Text.Json.Serialization;

namespace SystemTextJsonSamples
{
    public class DateTimeOffsetNullHandlingConverter : 
JsonConverter<DateTimeOffset>
    {
        public override DateTimeOffset Read(
            ref Utf8JsonReader reader,
            Type typeToConvert,
            JsonSerializerOptions options) =>
        reader.TokenType == JsonTokenType.Null
            ? default
            : reader.GetDateTimeOffset();

        public override void Write(
            Utf8JsonWriter writer,
            DateTimeOffset dateTimeValue,
            JsonSerializerOptions options) =>
        writer.WriteStringValue(dateTimeValue);
    }
}
```

Register this custom converter by [using an attribute on the property](#) or by [adding the converter](#) to the `Converters` collection.

Note: The preceding converter **handles null values differently** than `Newtonsoft.Json` does for POCOs that specify default values. For example, suppose the following code represents your target object:

C#

```
public class WeatherForecastWithDefault
{
    public WeatherForecastWithDefault()
    {
```

```
        Date = DateTimeOffset.Parse("2001-01-01");
        Summary = "No summary";
    }
    public DateTimeOffset Date { get; set; }
    public int TemperatureCelsius { get; set; }
    public string Summary { get; set; }
}
```

And suppose the following JSON is deserialized by using the preceding converter:

JSON

```
{
    "Date": null,
    "TemperatureCelsius": 25,
    "Summary": null
}
```

After deserialization, the `Date` property has 1/1/0001 (`default(DateTimeOffset)`), that is, the value set in the constructor is overwritten. Given the same POCO and JSON, `Newtonsoft.Json` deserialization would leave 1/1/2001 in the `Date` property.

Deserialize to immutable classes and structs

`Newtonsoft.Json` can deserialize to immutable classes and structs because it can use constructors that have parameters.

In `System.Text.Json`, use the `[JsonConstructor]` attribute to specify use of a parameterized constructor. Records in C# 9 are also immutable and are supported as deserialization targets. For more information, see [Immutable types and Records](#).

Required properties

In `Newtonsoft.Json`, you specify that a property is required by setting `Required` on the `[JsonProperty]` attribute. `Newtonsoft.Json` throws an exception if no value is received in the JSON for a property marked as required.

Specify date format

`Newtonsoft.Json` provides several ways to control how properties of `DateTime` and `DateTimeOffset` types are serialized and deserialized:

- The `DateTimeZoneHandling` setting can be used to serialize all `DateTime` values as UTC dates.
- The `DateFormatString` setting and `DateTime` converters can be used to customize the format of date strings.

`System.Text.Json` supports ISO 8601-1:2019, including the RFC 3339 profile. This format is widely adopted, unambiguous, and makes round trips precisely. To use any other format, create a custom converter. For example, the following converters serialize and deserialize JSON that uses Unix epoch format with or without a time zone offset (values such as `/Date(1590863400000-0700)/` or `/Date(1590863400000)/`):

C#

```
sealed class UnixEpochDateTimeOffsetConverter :  
JsonConverter<DateTimeOffset>  
{  
    static readonly DateTimeOffset s_epoch = new DateTimeOffset(1970, 1, 1,  
0, 0, 0, TimeSpan.Zero);  
    static readonly Regex s_regex = new Regex("^/Date\\\"(([+-]*)((\\d{2})\\(\\d{2}\\))\\)\\/$", RegexOptions.CultureInvariant);  
  
    public override DateTimeOffset Read(ref Utf8JsonReader reader, Type  
typeToConvert, JsonSerializerOptions options)  
    {  
        string formatted = reader.GetString()!;  
        Match match = s_regex.Match(formatted);  
  
        if (  
            !match.Success  
            || !long.TryParse(match.Groups[1].Value,  
System.Globalization.NumberStyles.Integer, CultureInfo.InvariantCulture, out  
long unixTime)  
            || !int.TryParse(match.Groups[3].Value,  
System.Globalization.NumberStyles.Integer, CultureInfo.InvariantCulture, out  
int hours)  
            || !int.TryParse(match.Groups[4].Value,  
System.Globalization.NumberStyles.Integer, CultureInfo.InvariantCulture, out  
int minutes))  
        {  
            throw new JsonException();  
        }  
  
        int sign = match.Groups[2].Value[0] == '+' ? 1 : -1;  
        TimeSpan utcOffset = new TimeSpan(hours * sign, minutes * sign, 0);  
  
        return s_epoch.AddMilliseconds(unixTime).ToOffset(utcOffset);  
    }  
  
    public override void Write(Utf8JsonWriter writer, DateTimeOffset value,  
JsonSerializerOptions options)  
    {
```

```
        long unixTime = Convert.ToInt64((value -  
s_epoch).TotalMilliseconds);  
        TimeSpan utcOffset = value.Offset;  
  
        string formatted = string.Create(CultureInfo.InvariantCulture,  
$/Date({unixTime}{(utcOffset >= TimeSpan.Zero ? "+" : "-")}  
{utcOffset:hhmm})/");  
  
        writer.WriteStringValue(formatted);  
    }  
}
```

C#

```
sealed class UnixEpochDateTimeConverter : JsonConverter<DateTime>
{
    static readonly DateTime s_epoch = new DateTime(1970, 1, 1, 0, 0, 0);
    static readonly Regex s_regex = new Regex("^/Date\\\"(([+-]*/\\d+)\\\")/$",
RegexOptions.CultureInvariant);

    public override DateTime Read(ref Utf8JsonReader reader, Type typeToConvert, JsonSerializerOptions options)
    {
        string formatted = reader.GetString()!;
        Match match = s_regex.Match(formatted);

        if (
            !match.Success
            || !long.TryParse(match.Groups[1].Value,
System.Globalization.NumberStyles.Integer, CultureInfo.InvariantCulture, out
long unixTime))
        {
            throw new JsonException();
        }

        return s_epoch.AddMilliseconds(unixTime);
    }

    public override void Write(Utf8JsonWriter writer, DateTime value,
JsonSerializerOptions options)
    {
        long unixTime = Convert.ToInt64((value -
s_epoch).TotalMilliseconds);

        string formatted = string.Create(CultureInfo.InvariantCulture,
$"/Date({unixTime})/");
        writer.WriteStringValue(formatted);
    }
}
```

For more information, see [DateTime and DateTimeOffset support in System.Text.Json](#).

Callbacks

`Newtonsoft.Json` lets you execute custom code at several points in the serialization or deserialization process:

- `OnDeserializing` (when beginning to deserialize an object)
- `OnDeserialized` (when finished deserializing an object)
- `OnSerializing` (when beginning to serialize an object)
- `OnSerialized` (when finished serializing an object)

`System.Text.Json` exposes the same notifications during serialization and deserialization. To use them, implement one or more of the following interfaces from the [System.Text.Json.Serialization](#) namespace:

- [IJsonOnDeserializing](#)
- [IJsonOnDeserialized](#)
- [IJsonOnSerializing](#)
- [IJsonOnSerialized](#)

Here's an example that checks for a null property and writes messages at start and end of serialization and deserialization:

C#

```
using System.Text.Json;
using System.Text.Json.Serialization;

namespace Callbacks
{
    public class WeatherForecast :
        IJsonOnDeserializing, IJsonOnDeserialized,
        IJsonOnSerializing, IJsonOnSerialized
    {
        public DateTime Date { get; set; }
        public int TemperatureCelsius { get; set; }
        public string? Summary { get; set; }

        void IJsonOnDeserializing.OnDeserializing() =>
Console.WriteLine("\nBegin deserializing");
        void IJsonOnDeserialized.OnDeserialized()
        {
            Validate();
            Console.WriteLine("Finished deserializing");
        }
        void IJsonOnSerializing.OnSerializing()
        {
            Console.WriteLine("Begin serializing");
            Validate();
        }
    }
}
```

```

        void IJsonOnSerialized.OnSerialized() => Console.WriteLine("Finished
serializing");

        private void Validate()
        {
            if (Summary is null)
            {
                Console.WriteLine("The 'Summary' property is 'null'.");
            }
        }
    }

    public class Program
    {
        public static void Main()
        {
            var weatherForecast = new WeatherForecast
            {
                Date = DateTime.Parse("2019-08-01"),
                TemperatureCelsius = 25,
            };

            string jsonString = JsonSerializer.Serialize(weatherForecast);
            Console.WriteLine(jsonString);

            weatherForecast = JsonSerializer.Deserialize<WeatherForecast>
(jjsonString);
            Console.WriteLine($"Date={weatherForecast?.Date}");
            Console.WriteLine($"TemperatureCelsius=
{weatherForecast?.TemperatureCelsius}");
            Console.WriteLine($"Summary={weatherForecast?.Summary}");
        }
    }
}

// output:
//Begin serializing
//The 'Summary' property is 'null'.
//Finished serializing
//{"Date":"2019-08-01T00:00:00","TemperatureCelsius":25,"Summary":null}

//Begin deserializing
//The 'Summary' property is 'null'.
//Finished deserializing
//Date=8/1/2019 12:00:00 AM
//TemperatureCelsius = 25
//Summary=

```

The `OnDeserializing` code doesn't have access to the new POCO instance. To manipulate the new POCO instance at the start of deserialization, put that code in the POCO constructor.

Non-public property setters and getters

`Newtonsoft.Json` can use private and internal property setters and getters via the `JsonProperty` attribute.

`System.Text.Json` supports private and internal property setters and getters via the `[JsonInclude]` attribute. For sample code, see [Non-public property accessors](#).

Populate existing objects

The `JsonConvert.PopulateObject` method in `Newtonsoft.Json` deserializes a JSON document to an existing instance of a class, instead of creating a new instance. `System.Text.Json` always creates a new instance of the target type by using the default public parameterless constructor. Custom converters can deserialize to an existing instance.

Reuse rather than replace properties

Starting in .NET 8, `System.Text.Json` supports reusing initialized properties rather than replacing them. There are some differences in behavior, which you can read about in the [API proposal ↗](#).

For more information, see [Populate initialized properties](#).

Populate properties without setters

Starting in .NET 8, `System.Text.Json` supports populating properties, including those that don't have a setter. For more information, see [Populate initialized properties](#).

Snake case naming policy

`System.Text.Json` includes a built-in naming policy for snake case. However, there are some behavior differences with `Newtonsoft.Json` for some inputs. The following table shows some of these differences when converting input using the `JsonNamingPolicy.SnakeCaseLower` policy.

| Input | Newtonsoft.Json result | System.Text.Json result |
|-----------------|-------------------------------|--------------------------------|
| "AB1" | "a_b1" | "ab1" |
| "SHA512Managed" | "sh_a512_managed" | "sha512_managed" |
| "abc123DEF456" | "abc123_de_f456" | "abc123_def456" |
| "KEBAB-CASE" | "keba_b-_case" | "kebab-case" |

System.Runtime.Serialization attributes

`System.Runtime.Serialization` attributes such as `DataContractAttribute`, `DataMemberAttribute`, and `IgnoreDataMemberAttribute` let you define a *data contract*. A data contract is a formal agreement between a service and a client that abstractly describes the data to be exchanged. The data contract precisely defines which properties are serialized for exchange.

`System.Text.Json` doesn't have built-in support for these attributes. However, starting in .NET 7, you can use a [custom type resolver](#) to add support. For a sample, see [ZCS.DataContractResolver](#).

Octal numbers

`Newtonsoft.Json` treats numbers with a leading zero as octal numbers. `System.Text.Json` doesn't allow leading zeroes because the [RFC 8259](#) specification doesn't allow them.

Handle missing members

If the JSON that's being deserialized includes properties that are missing in the target type, `Newtonsoft.Json` can be configured to throw exceptions. By default, `System.Text.Json` ignores extra properties in the JSON, except when you use the [\[JsonExtensionData\] attribute](#).

In .NET 8 and later versions, you can set your preference for whether to skip or disallow unmapped JSON properties using one of the following means:

- Apply the `JsonUnmappedMemberHandlingAttribute` attribute to the type you're deserializing to.
- To set your preference globally, set the `JsonSerializerOptions.UnmappedMemberHandling` property. Or, for source generation, set the `JsonSourceGenerationOptionsAttribute.UnmappedMemberHandling` property and apply the attribute to your `JsonSerializerContext` class.
- Customize the `JsonTypeInfo.UnmappedMemberHandling` property.

JsonObjectAttribute

`Newtonsoft.Json` has an attribute, `JsonObjectAttribute`, that can be applied at the *type level* to control which members are serialized, how `null` values are handled, and whether all members are required. `System.Text.Json` has no equivalent attribute that can

be applied on a type. For some behaviors, such as `null` value handling, you can either configure the same behavior on the global `JsonSerializerOptions` or individually on each property.

Consider the following example that uses `Newtonsoft.Json.JsonObjectAttribute` to specify that all `null` properties should be ignored:

```
C#
```

```
[JsonObject(ItemNullValueHandling = NullValueHandling.Ignore)]
public class Person { ... }
```

In `System.Text.Json`, you can set the behavior [for all types and properties](#):

```
C#
```

```
JsonSerializerOptions options = new()
{
    DefaultIgnoreCondition = JsonIgnoreCondition.WhenWritingNull
};

string json = JsonSerializer.Serialize<Person>(person, options);
```

Or you can set the behavior [on each property separately](#):

```
C#
```

```
public class Person
{
    [JsonIgnore(Condition = JsonIgnoreCondition.WhenWritingNull)]
    public string? Name { get; set; }

    [JsonIgnore(Condition = JsonIgnoreCondition.WhenWritingNull)]
    public int? Age { get; set; }
}
```

Next, consider the following example that uses `Newtonsoft.Json.JsonObjectAttribute` to specify that all member properties must be present in the JSON:

```
C#
```

```
[JsonObject(ItemRequired = Required.Always)]
public class Person { ... }
```

You can achieve the same behavior in `System.Text.Json` by adding the C# `required` modifier or the `JsonPropertyAttribute` to *each property*. For more information, see

Required properties.

C#

```
public class Person
{
    [JsonRequired]
    public string? Name { get; set; }

    public required int? Age { get; set; }
}
```

TraceWriter

`Newtonsoft.Json` lets you debug by using a `TraceWriter` to view logs that are generated by serialization or deserialization. `System.Text.Json` doesn't do logging.

JsonDocument and JsonElement compared to JToken (like JObject, JArray)

`System.Text.Json.JsonDocument` provides the ability to parse and build a **read-only** Document Object Model (DOM) from existing JSON payloads. The DOM provides random access to data in a JSON payload. The JSON elements that compose the payload can be accessed via the `JsonElement` type. The `JsonElement` type provides APIs to convert JSON text to common .NET types. `JsonDocument` exposes a `RootElement` property.

Starting in .NET 6, you can parse and build a **mutable** DOM from existing JSON payloads by using the `JsonNode` type and other types in the `System.Text.Json.Nodes` namespace. For more information, see [Use JsonNode](#).

JsonDocument is IDisposable

`JsonDocument` builds an in-memory view of the data into a pooled buffer. Therefore, unlike `JObject` or `JArray` from `Newtonsoft.Json`, the `JsonDocument` type implements `IDisposable` and needs to be used inside a using block. For more information, see [JsonDocument is IDisposable](#).

JsonDocument is read-only

The `System.Text.Json` DOM can't add, remove, or modify JSON elements. It's designed this way for performance and to reduce allocations for parsing common JSON payload sizes (that is, < 1 MB).

JsonElement is a union struct

`JsonDocument` exposes the `RootElement` as a property of type `JsonElement`, which is a union struct type that encompasses any JSON element. `Newtonsoft.Json` uses dedicated hierarchical types like `JObject`, `JArray`, `JToken`, and so forth. `JsonElement` is what you can search and enumerate over, and you can use `JsonElement` to materialize JSON elements into .NET types.

Starting in .NET 6, you can use `JsonNode` type and types in the `System.Text.Json.Nodes` namespace that correspond to `JObject`, `JArray`, and `JToken`. For more information, see [Use JsonNode](#).

How to search a JsonDocument and JsonElement for sub-elements

Searches for JSON tokens using `JObject` or `JArray` from `Newtonsoft.Json` tend to be relatively fast because they're lookups in some dictionary. By comparison, searches on `JsonElement` require a sequential search of the properties and hence are relatively slow (for example when using `TryGetProperty`). `System.Text.Json` is designed to minimize initial parse time rather than lookup time. For more information, see [How to search a JsonDocument and JsonElement for sub-elements](#).

Utf8JsonReader compared to JsonTextReader

`System.Text.Json.Utf8JsonReader` is a high-performance, low allocation, forward-only reader for UTF-8 encoded JSON text, read from a `ReadOnlySpan<byte>` or `ReadOnlySequence<byte>`. The `Utf8JsonReader` is a low-level type that can be used to build custom parsers and deserializers.

Utf8JsonReader is a ref struct

The `JsonTextReader` in `Newtonsoft.Json` is a class. The `Utf8JsonReader` type differs in that it's a *ref struct*. For more information, see [ref struct limitations for Utf8JsonReader](#).

Read null values into nullable value types

`Newtonsoft.Json` provides APIs that return `Nullable<T>`, such as `ReadAsBoolean`, which handles a `Null TokenType` for you by returning a `bool?`. The built-in `System.Text.Json` APIs return only non-nullable value types. For more information, see [Read null values into nullable value types](#).

Multi-targeting

If you need to continue to use `Newtonsoft.Json` for certain target frameworks, you can multi-target and have two implementations. However, this is not trivial and would require some `#ifdefs` and source duplication. One way to share as much code as possible is to create a `ref struct` wrapper around `Utf8JsonReader` and `Newtonsoft.Json.JsonTextReader`. This wrapper would unify the public surface area while isolating the behavioral differences. This lets you isolate the changes mainly to the construction of the type, along with passing the new type around by reference. This is the pattern that the [Microsoft.Extensions.DependencyModel](#) library follows:

- [UnifiedJsonReader.JsonTextReader.cs](#)
- [UnifiedJsonReader.Utf8JsonReader.cs](#)

Utf8JsonWriter compared to JsonTextWriter

`System.Text.Json.Utf8JsonWriter` is a high-performance way to write UTF-8 encoded JSON text from common .NET types like `String`, `Int32`, and `DateTime`. The writer is a low-level type that can be used to build custom serializers.

Write raw values

The `Newtonsoft.Json` `WriteRawValue` method writes raw JSON where a value is expected. `System.Text.Json` has a direct equivalent: `Utf8JsonWriter.WriteRawValue`. For more information, see [Write raw JSON](#).

Customize JSON format

`JsonTextWriter` includes the following settings, for which `Utf8JsonWriter` has no equivalent:

- `Indentation` - Specifies how many characters to indent. `Utf8JsonWriter` always indents by 2 characters.
- `IndentChar` - Specifies the character to use for indentation. `Utf8JsonWriter` always uses whitespace.

- `QuoteChar` ↴ - Specifies the character to use to surround string values.
`Utf8JsonWriter` always uses double quotes.
- `QuoteName` ↴ - Specifies whether or not to surround property names with quotes.
`Utf8JsonWriter` always surrounds them with quotes.

There are no workarounds that would let you customize the JSON produced by `Utf8JsonWriter` in these ways.

Write Timespan, Uri, or char values

`JsonTextWriter` provides `WriteValue` methods for `TimeSpan` ↴, `Uri` ↴, and `char` ↴ values. `Utf8JsonWriter` doesn't have equivalent methods. Instead, format these values as strings (by calling `ToString()`, for example) and call `WriteStringValue`.

Multi-targeting

If you need to continue to use `Newtonsoft.Json` for certain target frameworks, you can multi-target and have two implementations. However, this is not trivial and would require some `#ifdefs` and source duplication. One way to share as much code as possible is to create a wrapper around `Utf8JsonWriter` and `Newtonsoft.JsonTextWriter`. This wrapper would unify the public surface area while isolating the behavioral differences. This lets you isolate the changes mainly to the construction of the type. `Microsoft.Extensions.DependencyModel` ↴ library follows:

- `UnifiedJsonWriter.JsonTextWriter.cs` ↴
- `UnifiedJsonWriter.Utf8JsonWriter.cs` ↴

TypeNameHandling.All not supported

The decision to exclude `TypeNameHandling.All`-equivalent functionality from `System.Text.Json` was intentional. Allowing a JSON payload to specify its own type information is a common source of vulnerabilities in web applications. In particular, configuring `Newtonsoft.Json` with `TypeNameHandling.All` allows the remote client to embed an entire executable application within the JSON payload itself, so that during deserialization the web application extracts and runs the embedded code. For more information, see [Friday the 13th JSON attacks PowerPoint](#) ↴ and [Friday the 13th JSON attacks details](#) ↴.

JSON Path queries not supported

The `JsonDocument` DOM doesn't support querying by using JSON Path[↗].

In a `JsonNode` DOM, each `JsonNode` instance has a `GetPath` method that returns a path to that node. But there is no built-in API to handle queries based on JSON Path query strings.

For more information, see the [dotnet/runtime #31068 GitHub issue[↗]](#).

Some limits not configurable

`System.Text.Json` sets limits that can't be changed for some values, such as the maximum token size in characters (166 MB) and in base 64 (125 MB). For more information, see [JsonConstants in the source code[↗]](#) and GitHub issue [dotnet/runtime #39953[↗]](#).

NaN, Infinity, -Infinity

Newtonsoft parses `NaN`, `Infinity`, and `-Infinity` JSON string tokens. With `System.Text.Json`, use `JsonNumberHandling.AllowNamedFloatingPointLiterals`. For information about how to use this setting, see [Allow or write numbers in quotes](#).

Additional resources

- [System.Text.Json overview](#)
- [How to serialize and deserialize JSON](#)

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

How to instantiate JsonSerializerOptions instances with System.Text.Json

Article • 05/23/2023

This article explains how to avoid performance problems when you use [JsonSerializerOptions](#). It also shows how to use the parameterized constructors that are available.

Reuse JsonSerializerOptions instances

If you use `JsonSerializerOptions` repeatedly with the same options, don't create a new `JsonSerializerOptions` instance each time you use it. Reuse the same instance for every call. This guidance applies to code you write for custom converters and when you call `JsonSerializer.Serialize` or `JsonSerializer.Deserialize`. It's safe to use the same instance across multiple threads. The metadata caches on the options instance are thread-safe, and the instance is immutable after the first serialization or deserialization.

The `JsonSerializerOptions.Default` property

If the instance of `JsonSerializerOptions` that you need to use is the default instance (has all of the default settings and the default converters), use the `JsonSerializerOptions.Default` property rather than creating an options instance. For more information, see [Use default system converter](#).

Copy JsonSerializerOptions

There is a [JsonSerializerOptions constructor](#) that lets you create a new instance with the same options as an existing instance, as shown in the following example:

```
C#  
  
using System.Text.Json;  
  
namespace CopyOptions  
{  
    public class Forecast  
    {  
        public DateTime Date { get; init; }  
        public int TemperatureC { get; set; }  
        public string? Summary { get; set; }  
    };  
}
```

```

public class Program
{
    public static void Main()
    {
        Forecast forecast = new()
        {
            Date = DateTime.Now,
            TemperatureC = 40,
            Summary = "Hot"
        };

        JsonSerializerOptions options = new()
        {
            WriteIndented = true
        };

        JsonSerializerOptions optionsCopy = new(options);
        string forecastJson =
            JsonSerializer.Serialize<Forecast>(forecast, optionsCopy);

        Console.WriteLine($"Output JSON:\n{forecastJson}");
    }
}

// Produces output like the following example:
// 
//Output JSON:
//{
//    "Date": "2020-10-21T15:40:06.8998502-07:00",
//    "TemperatureC": 40,
//    "Summary": "Hot"
//}

```

The metadata cache of the existing `JsonSerializerOptions` instance isn't copied to the new instance. So using this constructor is not the same as reusing an existing instance of `JsonSerializerOptions`.

Web defaults for JsonSerializerOptions

The following options have different defaults for web apps:

- `PropertyNameCaseInsensitive` = `true`
- `JsonNamingPolicy` = `CamelCase`
- `NumberHandling` = `AllowReadingFromString`

The [JsonSerializerOptions constructor](#) lets you create a new instance with the default options that ASP.NET Core uses for web apps, as shown in the following example:

C#

```
using System.Text.Json;

namespace OptionsDefaults
{
    public class Forecast
    {
        public DateTime? Date { get; init; }
        public int TemperatureC { get; set; }
        public string? Summary { get; set; }
    };

    public class Program
    {
        public static void Main()
        {
            Forecast forecast = new()
            {
                Date = DateTime.Now,
                TemperatureC = 40,
                Summary = "Hot"
            };

            JsonSerializerOptions options = new(JsonSerializerDefaults.Web)
            {
                WriteIndented = true
            };

            Console.WriteLine(
                $"PropertyNameCaseInsensitive:
{options.PropertyNameCaseInsensitive}");
            Console.WriteLine(
                $"JsonNamingPolicy: {options.JsonNamingPolicy}");
            Console.WriteLine(
                $"NumberHandling: {options.NumberHandling}");

            string forecastJson = JsonSerializer.Serialize<Forecast>
(forecast, options);
            Console.WriteLine($"Output JSON:\n{forecastJson}");

            Forecast? forecastDeserialized =
                JsonSerializer.Deserialize<Forecast>(forecastJson, options);

            Console.WriteLine($"Date: {forecastDeserialized?.Date}");
            Console.WriteLine($"TemperatureC:
{forecastDeserialized?.TemperatureC}");
            Console.WriteLine($"Summary: {forecastDeserialized?.Summary}");
        }
    }

    // Produces output like the following example:
    //
}
```

```
//PropertyNameCaseInsensitive: True
//JsonNamingPolicy: System.Text.Json.JsonCamelCaseNamingPolicy
//NumberHandling: AllowReadingFromString
//Output JSON:
//{
//  "date": "2020-10-21T15:40:06.9040831-07:00",
//  "temperatureC": 40,
//  "summary": "Hot"
//}
//Date: 10 / 21 / 2020 3:40:06 PM
//TemperatureC: 40
//Summary: Hot
```

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

How to enable case-insensitive property name matching with System.Text.Json

Article • 05/26/2023

In this article, you will learn how to enable case-insensitive property name matching with the `System.Text.Json` namespace.

Case-insensitive property matching

By default, deserialization looks for case-sensitive property name matches between JSON and the target object properties. To change that behavior, set `JsonSerializerOptions.PropertyNameCaseInsensitive` to `true`:

ⓘ Note

The web default is case-insensitive.

C#

```
var options = new JsonSerializerOptions
{
    PropertyNameCaseInsensitive = true
};
var weatherForecast = JsonSerializer.Deserialize<WeatherForecast>
(jsonString, options);
```

Here's example JSON with camel case property names. It can be deserialized into the following type that has Pascal case property names.

JSON

```
{  
    "date": "2019-08-01T00:00:00-07:00",  
    "temperatureCelsius": 25,  
    "summary": "Hot",  
}
```

C#

```
public class WeatherForecast  
{  
    public DateTimeOffset Date { get; set; }
```

```
public int TemperatureCelsius { get; set; }
public string? Summary { get; set; }
}
```

See also

- [System.Text.Json overview](#)
- [How to serialize and deserialize JSON](#)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

How to preserve references and handle or ignore circular references in System.Text.Json

Article • 05/26/2023

This article shows how to preserve references and handle or ignore circular references while using System.Text.Json to serialize and deserialize JSON in .NET

Preserve references and handle circular references

To preserve references and handle circular references, set [ReferenceHandler](#) to `Preserve`. This setting causes the following behavior:

- On serialize:

When writing complex types, the serializer also writes metadata properties (`$id`, `$values`, and `$ref`).

- On deserialize:

Metadata is expected (although not mandatory), and the deserializer tries to understand it.

The following code illustrates use of the `Preserve` setting.

C#

```
using System.Text.Json;
using System.Text.Json.Serialization;

namespace PreserveReferences
{
    public class Employee
    {
        public string? Name { get; set; }
        public Employee? Manager { get; set; }
        public List<Employee>? DirectReports { get; set; }
    }

    public class Program
    {
        public static void Main()
```

```
    {
        Employee tyler = new()
        {
            Name = "Tyler Stein"
        };

        Employee adrian = new()
        {
            Name = "Adrian King"
        };

        tyler.DirectReports = new List<Employee> { adrian };
        adrian.Manager = tyler;

        JsonSerializerOptions options = new()
        {
            ReferenceHandler = ReferenceHandler.Preserve,
            WriteIndented = true
        };

        string tylerJson = JsonSerializer.Serialize(tyler, options);
        Console.WriteLine($"Tyler serialized:\n{tylerJson}");

        Employee? tylerDeserialized =
            JsonSerializer.Deserialize<Employee>(tylerJson, options);

        Console.WriteLine(
            "Tyler is manager of Tyler's first direct report: ");
        Console.WriteLine(
            tylerDeserialized?.DirectReports?[0].Manager ==
tylerDeserialized);
    }
}

// Produces output like the following example:
// 
//Tyler serialized:
//{
//  "$id": "1",
//  "Name": "Tyler Stein",
//  "Manager": null,
//  "DirectReports": [
//    {
//      "$id": "2",
//      "$values": [
//        {
//          "$id": "3",
//          "Name": "Adrian King",
//          "Manager": {
//            "$ref": "1"
//          },
//          "DirectReports": null
//        }
//      ]
//    }
//  ]
//}
```

```
//}  
//Tyler is manager of Tyler's first direct report:  
//True
```

This feature can't be used to preserve value types or immutable types. On deserialization, the instance of an immutable type is created after the entire payload is read. So it would be impossible to deserialize the same instance if a reference to it appears within the JSON payload.

For value types, immutable types, and arrays, no reference metadata is serialized. On deserialization, an exception is thrown if `$ref` or `$id` is found. However, value types ignore `$id` (and `$values` in the case of collections) to make it possible to deserialize payloads that were serialized by using Newtonsoft.Json. Newtonsoft.Json does serialize metadata for such types.

To determine if objects are equal, `System.Text.Json` uses [ReferenceEqualityComparer.Instance](#), which uses reference equality (`Object.ReferenceEquals(Object, Object)`) instead of value equality (`Object.Equals(Object)`) when comparing two object instances.

For more information about how references are serialized and deserialized, see [ReferenceHandler.Preserve](#).

The [ReferenceResolver](#) class defines the behavior of preserving references on serialization and deserialization. Create a derived class to specify custom behavior. For an example, see [GuidReferenceResolver](#).

Persist reference metadata across multiple serialization and deserialization calls

By default, reference data is only cached for each call to `Serialize` or `Deserialize`. To persist references from one `Serialize/Deserialize` call to another one, root the `ReferenceResolver` instance in the call site of `Serialize/Deserialize`. The following code shows an example for this scenario:

- You have a list of `Employee` objects and you have to serialize each one individually.
- You want to take advantage of the references saved in the resolver for the `ReferenceHandler`.

Here is the `Employee` class:

C#

```
public class Employee
{
    public string? Name { get; set; }
    public Employee? Manager { get; set; }
    public List<Employee>? DirectReports { get; set; }
}
```

A class that derives from [ReferenceResolver](#) stores the references in a dictionary:

C#

```
class MyReferenceResolver : ReferenceResolver
{
    private uint _referenceCount;
    private readonly Dictionary<string, object> _referenceIdToObjectMap =
    new ();
    private readonly Dictionary<object, string> _objectToReferenceIdMap =
    new (ReferenceEqualityComparer.Instance);

    public override void AddReference(string referenceId, object value)
    {
        if (!_referenceIdToObjectMap.TryAdd(referenceId, value))
        {
            throw new JsonException();
        }
    }

    public override string GetReference(object value, out bool
alreadyExists)
    {
        if (_objectToReferenceIdMap.TryGetValue(value, out string?
referenceId))
        {
            alreadyExists = true;
        }
        else
        {
            _referenceCount++;
            referenceId = _referenceCount.ToString();
            _objectToReferenceIdMap.Add(value, referenceId);
            alreadyExists = false;
        }
    }

    return referenceId;
}

public override object ResolveReference(string referenceId)
{
    if (!_referenceIdToObjectMap.TryGetValue(referenceId, out object?
value))
    {
        throw new JsonException();
    }
}
```

```
        }

        return value;
    }
}
```

A class that derives from [ReferenceHandler](#) holds an instance of [MyReferenceResolver](#) and creates a new instance only when needed (in a method named `Reset` in this example):

C#

```
class MyReferenceHandler : ReferenceHandler
{
    public MyReferenceHandler() => Reset();
    private ReferenceResolver? _rootedResolver;
    public override ReferenceResolver CreateResolver() => _rootedResolver!;
    public void Reset() => _rootedResolver = new MyReferenceResolver();
}
```

When the sample code calls the serializer, it uses a [JsonSerializerOptions](#) instance in which the [ReferenceHandler](#) property is set to an instance of [MyReferenceHandler](#). When you follow this pattern, be sure to reset the [ReferenceResolver](#) dictionary when you're finished serializing, to keep it from growing forever.

C#

```
var options = new JsonSerializerOptions();
options.WriteIndented = true;
var myReferenceHandler = new MyReferenceHandler();
options.ReferenceHandler = myReferenceHandler;

string json;
foreach (Employee emp in employees)
{
    json = JsonSerializer.Serialize(emp, options);
    DoSomething(json);
}

// Reset after serializing to avoid out of bounds memory growth in the
// resolver.
myReferenceHandler.Reset();
```

Ignore circular references

Instead of handling circular references, you can ignore them. To ignore circular references, set [ReferenceHandler](#) to [IgnoreCycles](#). The serializer sets circular reference properties to `null`, as shown in the following example:

C#

```
using System.Text.Json;
using System.Text.Json.Serialization;

namespace SerializeIgnoreCycles
{
    public class Employee
    {
        public string? Name { get; set; }
        public Employee? Manager { get; set; }
        public List<Employee>? DirectReports { get; set; }
    }

    public class Program
    {
        public static void Main()
        {
            Employee tyler = new()
            {
                Name = "Tyler Stein"
            };

            Employee adrian = new()
            {
                Name = "Adrian King"
            };

            tyler.DirectReports = new List<Employee> { adrian };
            adrian.Manager = tyler;

            JsonSerializerOptions options = new()
            {
                ReferenceHandler = ReferenceHandler.IgnoreCycles,
                WriteIndented = true
            };

            string tylerJson = JsonSerializer.Serialize(tyler, options);
            Console.WriteLine($"Tyler serialized:\n{tylerJson}");

            Employee? tylerDeserialized =
                JsonSerializer.Deserialize<Employee>(tylerJson, options);

            Console.WriteLine(
                "Tyler is manager of Tyler's first direct report: ");
            Console.WriteLine(
                tylerDeserialized?.DirectReports?[0]?.Manager ==
            tylerDeserialized);
        }
    }
}
```

```
        }

    // Produces output like the following example:
    //
    //Tyler serialized:
    //{
    //  "Name": "Tyler Stein",
    //  "Manager": null,
    //  "DirectReports": [
    //    {
    //      "Name": "Adrian King",
    //      "Manager": null,
    //      "DirectReports": null
    //    }
    //  ]
    //}
    //Tyler is manager of Tyler's first direct report:
    //False
```

In the preceding example, `Manager` under `Adrian King` is serialized as `null` to avoid the circular reference. This behavior has the following advantages over `ReferenceHandler.Preserve`:

- It decreases payload size.
- It creates JSON that is comprehensible for serializers other than `System.Text.Json` and `Newtonsoft.Json`.

This behavior has the following disadvantages:

- Silent loss of data.
- Data can't make a round trip from JSON back to the source object.

See also

- [System.Text.Json overview](#)
- [How to serialize and deserialize JSON](#)

 Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For

.NET

.NET feedback

The .NET documentation is open source. Provide feedback [here](#).

 [Open a documentation issue](#)

more information, see [our contributor guide](#).

 [Provide product feedback](#)

How to serialize properties of derived classes with System.Text.Json

Article • 05/26/2023

In this article, you will learn how to serialize properties of derived classes with the `System.Text.Json` namespace.

Serialize properties of derived classes

Beginning with .NET 7, `System.Text.Json` supports polymorphic type hierarchy serialization and deserialization with attribute annotations.

| Attribute | Description |
|---------------------------------------|--|
| <code>JsonDerivedTypeAttribute</code> | When placed on a type declaration, indicates that the specified subtype should be opted into polymorphic serialization. It also exposes the ability to specify a type discriminator. |
| <code>JsonPolymorphicAttribute</code> | When placed on a type declaration, indicates that the type should be serialized polymorphically. It also exposes various options to configure polymorphic serialization and deserialization for that type. |

For example, suppose you have a `WeatherForecastBase` class and a derived class `WeatherForecastWithCity`:

```
C#  
  
[JsonDerivedType(typeof(WeatherForecastWithCity))]  
public class WeatherForecastBase  
{  
    public DateTimeOffset Date { get; set; }  
    public int TemperatureCelsius { get; set; }  
    public string? Summary { get; set; }  
}
```

```
C#  
  
public class WeatherForecastWithCity : WeatherForecastBase  
{  
    public string? City { get; set; }  
}
```

And suppose the type argument of the `Serialize< TValue >` method at compile time is

`WeatherForecastBase`:

C#

```
options = new JsonSerializerOptions
{
    WriteIndented = true
};
jsonString = JsonSerializer.Serialize<WeatherForecastBase>
(weatherForecastBase, options);
```

In this scenario, the `City` property is serialized because the `weatherForecastBase` object is actually a `WeatherForecastWithCity` object. This configuration enables polymorphic serialization for `WeatherForecastBase`, specifically when the runtime type is

`WeatherForecastWithCity`:

JSON

```
{
    "City": "Milwaukee",
    "Date": "2022-09-26T00:00:00-05:00",
    "TemperatureCelsius": 15,
    "Summary": "Cool"
}
```

While round-tripping of the payload as `WeatherForecastBase` is supported, it won't materialize as a run-time type of `WeatherForecastWithCity`. Instead, it will materialize as a run-time type of `WeatherForecastBase`:

C#

```
WeatherForecastBase value = JsonSerializer.Deserialize<WeatherForecastBase>
(""""
{
    "City": "Milwaukee",
    "Date": "2022-09-26T00:00:00-05:00",
    "TemperatureCelsius": 15,
    "Summary": "Cool"
}
""");
Console.WriteLine(value is WeatherForecastWithCity); // False
```

The following section describes how to add metadata to enable round-tripping of the derived type.

Polymorphic type discriminators

To enable polymorphic deserialization, you must specify a type discriminator for the derived class:

C#

```
[JsonDerivedType(typeof(WeatherForecastBase), typeDiscriminator: "base")]
[JsonDerivedType(typeof(WeatherForecastWithCity), typeDiscriminator:
"withCity")]
public class WeatherForecastBase
{
    public DateTimeOffset Date { get; set; }
    public int TemperatureCelsius { get; set; }
    public string? Summary { get; set; }
}

public class WeatherForecastWithCity : WeatherForecastBase
{
    public string? City { get; set; }
}
```

With the added metadata, specifically, the type discriminator, the serializer can serialize and deserialize the payload as the `WeatherForecastWithCity` type from its base type `WeatherForecastBase`. Serialization will emit JSON along with the type discriminator metadata:

C#

```
WeatherForecastBase weather = new WeatherForecastWithCity
{
    City = "Milwaukee",
    Date = new DateTimeOffset(2022, 9, 26, 0, 0, 0, TimeSpan.FromHours(-5)),
    TemperatureCelsius = 15,
    Summary = "Cool"
}
var json = JsonSerializer.Serialize<WeatherForecastBase>(weather, options);
Console.WriteLine(json);
// Sample output:
// {
//     "$type" : "withCity",
//     "City": "Milwaukee",
//     "Date": "2022-09-26T00:00:00-05:00",
//     "TemperatureCelsius": 15,
//     "Summary": "Cool"
// }
```

With the type discriminator, the serializer can deserialize the payload polymorphically as `WeatherForecastWithCity`:

C#

```
WeatherForecastBase value = JsonSerializer.Deserialize<WeatherForecastBase>(json);
Console.WriteLine(value is WeatherForecastWithCity); // True
```

ⓘ Note

The type discriminator must be placed at the start of the JSON object, grouped together with other metadata properties like `$id` and `$ref`.

VB

```
Dim value As WeatherForecastBase = JsonSerializer.Deserialize(json)
Console.WriteLine(value is WeatherForecastWithCity) // True
```

Mix and match type discriminator formats

Type discriminator identifiers are valid in either `string` or `int` forms, so the following is valid:

C#

```
[JsonDerivedType(typeof(WeatherForecastWithCity), 0)]
[JsonDerivedType(typeof(WeatherForecastWithTimeSeries), 1)]
[JsonDerivedType(typeof(WeatherForecastWithLocalNews), 2)]
public class WeatherForecastBase { }

var json = JsonSerializer.Serialize<WeatherForecastBase>(new
WeatherForecastWithTimeSeries());
Console.WriteLine(json);
// Sample output:
// {
//   "$type" : 1,
//   // Omitted for brevity...
// }
```

While the API supports mixing and matching type discriminator configurations, it's not recommended. The general recommendation is to use either all `string` type discriminators, all `int` type discriminators, or no discriminators at all. The following example shows how to mix and match type discriminator configurations:

C#

```
[JsonDerivedType(typeof(ThreeDimensionalPoint), typeDiscriminator: 3)]
[JsonDerivedType(typeof(FourDimensionalPoint), typeDiscriminator: "4d")]
public class BasePoint
{
    public int X { get; set; }
    public int Y { get; set; }
}

public class ThreeDimensionalPoint : BasePoint
{
    public int Z { get; set; }
}

public sealed class FourDimensionalPoint : ThreeDimensionalPoint
{
    public int W { get; set; }
}
```

In the preceding example, the `BasePoint` type doesn't have a type discriminator, while the `ThreeDimensionalPoint` type has an `int` type discriminator, and the `FourDimensionalPoint` has a `string` type discriminator.

ⓘ Important

For polymorphic serialization to work, the type of the serialized value should be that of the polymorphic base type. This includes using the base type as the generic type parameter when serializing root-level values, as the declared type of serialized properties, or as the collection element in serialized collections.

C#

```
using System.Text.Json;
using System.Text.Json.Serialization;

PerformRoundTrip<BasePoint>();
PerformRoundTrip<ThreeDimensionalPoint>();
PerformRoundTrip<FourDimensionalPoint>();

static void PerformRoundTrip<T>() where T : BasePoint, new()
{
    var json = JsonSerializer.Serialize<BasePoint>(new T());
    Console.WriteLine(json);

    BasePoint? result = JsonSerializer.Deserialize<BasePoint>(json);
    Console.WriteLine($"result is {typeof(T)}; // {result is T}");
    Console.WriteLine();
```

```
}

// Sample output:
// { "X": 541, "Y": 503 }
// result is BasePoint; // True
//
// { "$type": 3, "Z": 399, "X": 835, "Y": 78 }
// result is ThreeDimensionalPoint; // True
//
// { "$type": "4d", "W": 993, "Z": 427, "X": 508, "Y": 741 }
// result is FourDimensionalPoint; // True
```

Customize the type discriminator name

The default property name for the type discriminator is `$type`. To customize the property name, use the [JsonPolymorphicAttribute](#) as shown in the following example:

C#

```
[JsonPolymorphic(TypeDiscriminatorPropertyName = "$discriminator")]
[JsonDerivedType(typeof(ThreeDimensionalPoint), typeDiscriminator: "3d")]
public class BasePoint
{
    public int X { get; set; }
    public int Y { get; set; }
}

public sealed class ThreeDimensionalPoint : BasePoint
{
    public int Z { get; set; }
}
```

In the preceding code, the `JsonPolymorphic` attribute configures the `TypeDiscriminatorPropertyName` to the `"$discriminator"` value. With the type discriminator name configured, the following example shows the `ThreeDimensionalPoint` type serialized as JSON:

C#

```
BasePoint point = new ThreeDimensionalPoint { X = 1, Y = 2, Z = 3 };
var json = JsonSerializer.Serialize<BasePoint>(point);
Console.WriteLine(json);
// Sample output:
// { "$discriminator": "3d", "X": 1, "Y": 2, "Z": 3 }
```

 Tip

Avoid using a `JsonPolymorphicAttribute.TypeDiscriminatorPropertyName` that conflicts with a property in your type hierarchy.

Handle unknown derived types

To handle unknown derived types, you must opt into such support using an annotation on the base type. Consider the following type hierarchy:

C#

```
[JsonDerivedType(typeof(ThreeDimensionalPoint))]
public class BasePoint
{
    public int X { get; set; }
    public int Y { get; set; }
}

public class ThreeDimensionalPoint : BasePoint
{
    public int Z { get; set; }
}

public class FourDimensionalPoint : ThreeDimensionalPoint
{
    public int W { get; set; }
}
```

Since the configuration does not explicitly opt-in support for `FourDimensionalPoint`, attempting to serialize instances of `FourDimensionalPoint` as `BasePoint` will result in a run-time exception:

C#

```
JsonSerializer.Serialize<BasePoint>(new FourDimensionalPoint()); // throws
NotSupportedException
```

You can change the default behavior by using the `JsonUnknownDerivedTypeHandling` enum, which can be specified as follows:

C#

```
[JsonPolymorphic(
    UnknownDerivedTypeHandling =
    JsonUnknownDerivedTypeHandling.FallBackToBaseType)]
[JsonDerivedType(typeof(ThreeDimensionalPoint))]
public class BasePoint
{
```

```

    public int X { get; set; }
    public int Y { get; set; }
}

public class ThreeDimensionalPoint : BasePoint
{
    public int Z { get; set; }
}

public class FourDimensionalPoint : ThreeDimensionalPoint
{
    public int W { get; set; }
}

```

Instead of falling back to the base type, you can use the `FallBackToNearestAncestor` setting to fall back to the contract of the nearest declared derived type:

C#

```

[JsonPolymorphic(
    UnknownDerivedTypeHandling =
    JsonUnknownDerivedTypeHandling.FallBackToNearestAncestor)]
[JsonDerivedType(typeof(BasePoint))]
public interface IPoint { }

public class BasePoint : IPoint { }

public class ThreeDimensionalPoint : BasePoint { }

```

With a configuration like the preceding example, the `ThreeDimensionalPoint` type will be serialized as `BasePoint`:

C#

```

// Serializes using the contract for BasePoint
JsonSerializer.Serialize<IPoint>(new ThreeDimensionalPoint());

```

However, falling back to the nearest ancestor admits the possibility of "diamond" ambiguity. Consider the following type hierarchy as an example:

C#

```

[JsonPolymorphic(
    UnknownDerivedTypeHandling =
    JsonUnknownDerivedTypeHandling.FallBackToNearestAncestor)]
[JsonDerivedType(typeof(BasePoint))]
[JsonDerivedType(typeof(IPointWithTimeSeries))]
public interface IPoint { }

```

```
public interface IPoIntWithTimeSeries : IPoInt { }

public class BasePoInt : IPoInt { }

public class BasePoIntWithTimeSeries : BasePoInt, IPoIntWithTimeSeries { }
```

In this case, the `BasePoIntWithTimeSeries` type could be serialized as either `BasePoInt` or `IPoIntWithTimeSeries` since they are both direct ancestors. This ambiguity will cause the [NotSupportedException](#) to be thrown when attempting to serialize an instance of `BasePoIntWithTimeSeries` as `IPoInt`.

C#

```
// throws NotSupportedException
JsonSerializer.Serialize<IPoInt>(new BasePoIntWithTimeSeries());
```

Configure polymorphism with the contract model

For use cases where attribute annotations are impractical or impossible (such as large domain models, cross-assembly hierarchies, or hierarchies in third-party dependencies), to configure polymorphism use the [contract model](#). The contract model is a set of APIs that can be used to configure polymorphism in a type hierarchy by creating a custom [DefaultJsonTypeInfoResolver](#) subclass that dynamically provides polymorphic configuration per type, as shown in the following example:

C#

```
public class PolymorphicTypeResolver : DefaultJsonTypeInfoResolver
{
    public override JsonTypeInfo GetTypeInfo(Type type,
JsonSerializerOptions options)
    {
        JsonTypeInfo jsonTypeInfo = base.GetTypeInfo(type, options);

        Type basePoIntType = typeof(BasePoInt);
        if (jsonTypeInfo.Type == basePoIntType)
        {
            jsonTypeInfo.PolymorphismOptions = new JsonPolymorphismOptions
            {
                TypeDiscriminatorPropertyName = "$point-type",
                IgnoreUnrecognizedTypeDiscriminators = true,
                UnknownDerivedTypeHandling =
JsonUnknownDerivedTypeHandling.FailSerialization,
                DerivedTypes =
            {

```

```
        new JsonDerivedType(typeof(ThreeDimensionalPoint),  
    "3d"),  
        new JsonDerivedType(typeof(FourDimensionalPoint), "4d")  
    }  
};  
  
return jsonTypeInfo;  
}  
}
```

Additional polymorphic serialization details

- Polymorphic serialization supports derived types that have been explicitly opted in via the [JsonDerivedTypeAttribute](#). Undeclared types will result in a run-time exception. The behavior can be changed by configuring the [JsonPolymorphicAttribute.UnknownDerivedTypeHandling](#) property.
- Polymorphic configuration specified in derived types is not inherited by polymorphic configuration in base types. The base type must be configured independently.
- Polymorphic hierarchies are supported for both `interface` and `class` types.
- Polymorphism using type discriminators is only supported for type hierarchies that use the default converters for objects, collections, and dictionary types.
- Polymorphism is supported in metadata-based source generation, but not fast-path source generation.

See also

- [System.Text.Json overview](#)
- [How to serialize and deserialize JSON](#)

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

How to use a JSON document object model in System.Text.Json

Article • 05/26/2023

This article shows how to use a [JSON document object model \(DOM\)](#) for random access to data in a JSON payload.

JSON DOM choices

Working with a DOM is an alternative to deserialization with [JsonSerializer](#) when:

- You don't have a type to deserialize into.
- The JSON you receive doesn't have a fixed schema and must be inspected to know what it contains.

`System.Text.Json` provides two ways to build a JSON DOM:

- [JsonDocument](#) provides the ability to build a read-only DOM by using `Utf8JsonReader`. The JSON elements that compose the payload can be accessed via the [JsonElement](#) type. The `JsonElement` type provides array and object enumerators along with APIs to convert JSON text to common .NET types. `JsonDocument` exposes a [RootElement](#) property. For more information, see [Use JsonDocument](#) later in this article.
- [JsonNode](#) and the classes that derive from it in the [System.Text.Json.Nodes](#) namespace provide the ability to create a mutable DOM. The JSON elements that compose the payload can be accessed via the [JsonNode](#), [JsonObject](#), [JsonArray](#), [JsonValue](#), and [JsonElement](#) types. For more information, see [Use JsonNode](#) later in this article.

Consider the following factors when choosing between `JsonDocument` and `JsonNode`:

- The `JsonNode` DOM can be changed after it's created. The `JsonDocument` DOM is immutable.
- The `JsonDocument` DOM provides faster access to its data.

Use `JsonNode`

The following example shows how to use `JsonNode` and the other types in the [System.Text.Json.Nodes](#) namespace to:

- Create a DOM from a JSON string
- Write JSON from a DOM.
- Get a value, object, or array from a DOM.

C#

```
using System.Text.Json;
using System.Text.Json.Nodes;

namespace JsonNodeFromStringExample;

public class Program
{
    public static void Main()
    {
        string jsonString =
@"{
    ""Date"": ""2019-08-01T00:00:00"",
    ""Temperature"": 25,
    ""Summary"": ""Hot"",
    ""DatesAvailable"": [
        ""2019-08-01T00:00:00"",
        ""2019-08-02T00:00:00""
    ],
    ""TemperatureRanges"": {
        ""Cold"": {
            ""High"": 20,
            ""Low"": -10
        },
        ""Hot"": {
            ""High"": 60,
            ""Low"": 20
        }
    }
};

// Create a JsonNode DOM from a JSON string.
JsonNode forecastNode = JsonNode.Parse(jsonString);

// Write JSON from a JsonNode
var options = new JsonSerializerOptions { WriteIndented = true };
Console.WriteLine(forecastNode!.ToString(options));
// output:
//{
//    "Date": "2019-08-01T00:00:00",
//    "Temperature": 25,
//    "Summary": "Hot",
//    "DatesAvailable": [
//        "2019-08-01T00:00:00",
//        "2019-08-02T00:00:00"
//    ],
//    "TemperatureRanges": {
//        "Cold": {
//            "High": 20,
//            "Low": -10
//        },
//        "Hot": {
//            "High": 60,
//            "Low": 20
//        }
//    }
}
```

```

        //      "High": 20,
        //      "Low": -10
        //    },
        //    "Hot": {
        //      "High": 60,
        //      "Low": 20
        //    }
        //}
      }

      // Get value from a JsonNode.
JsonNode temperatureNode = forecastNode!["Temperature"]!;
Console.WriteLine($"Type={temperatureNode.GetType()}");
Console.WriteLine($"JSON={temperatureNode.ToString()}");
//output:
//Type =
System.Text.Json.Nodes.JsonValue`1[System.Text.Json.JsonElement]
//JSON = 25

      // Get a typed value from a JsonNode.
int temperatureInt = (int)forecastNode!["Temperature"]!;
Console.WriteLine($"Value={temperatureInt}");
//output:
//Value=25

      // Get a typed value from a JsonNode by using GetValue<T>.
temperatureInt = forecastNode!["Temperature"]!.GetValue<int>();
Console.WriteLine($"TemperatureInt={temperatureInt}");
//output:
//Value=25

      // Get a JSON object from a JsonNode.
JsonNode temperatureRanges = forecastNode!["TemperatureRanges"]!;
Console.WriteLine($"Type={temperatureRanges.GetType()}");
Console.WriteLine($"JSON={temperatureRanges.ToString()}");
//output:
//Type = System.Text.Json.Nodes.JsonObject
//JSON = { "Cold":{ "High":20,"Low":-10}, "Hot":{ "High":60,"Low":20}
}

      // Get a JSON array from a JsonNode.
JsonNode datesAvailable = forecastNode!["DatesAvailable"]!;
Console.WriteLine($"Type={datesAvailable.GetType()}");
Console.WriteLine($"JSON={datesAvailable.ToString()}");
//output:
//datesAvailable Type = System.Text.Json.Nodes.JsonArray
//datesAvailable JSON =[{"2019-08-01T00:00:00", "2019-08-
02T00:00:00"}]

      // Get an array element value from a JSONArray.
JsonNode firstDateAvailable = datesAvailable[0]!;
Console.WriteLine($"Type={firstDateAvailable.GetType()}");
Console.WriteLine($"JSON={firstDateAvailable.ToString()}");
//output:
//Type =

```

```

System.Text.Json.Nodes.JsonValue`1[System.Text.Json.JsonElement]
//JSON = "2019-08-01T00:00:00"

    // Get a typed value by chaining references.
    int coldHighTemperature = (int)forecastNode["TemperatureRanges"]![
    ["Cold"]![["High"]!];
        Console.WriteLine($"TemperatureRanges.Cold.High={coldHighTemperature}");
        //output:
        //TemperatureRanges.Cold.High = 20

        // Parse a JSON array
        var datesNode = JsonNode.Parse(@"[\"2019-08-01T00:00:00\", \"2019-08-
02T00:00:00\"]");
        JsonNode firstDate = datesNode![0]!.GetValue<DateTime>();
        Console.WriteLine($"firstDate={ firstDate}");
        //output:
        //firstDate = "2019-08-01T00:00:00"
    }
}

```

Create a JsonNode DOM with object initializers and make changes

The following example shows how to:

- Create a DOM by using object initializers.
- Make changes to a DOM.

C#

```

using System.Text.Json;
using System.Text.Json.Nodes;

namespace JsonNodeFromObjectExample;

public class Program
{
    public static void Main()
    {
        // Create a new JsonObject using object initializers.
        var forecastObject = new JsonObject
        {
            ["Date"] = new DateTime(2019, 8, 1),
            ["Temperature"] = 25,
            ["Summary"] = "Hot",
            ["DatesAvailable"] = new JsonArray(
                new DateTime(2019, 8, 1), new DateTime(2019, 8, 2)),
            ["TemperatureRanges"] = new JsonObject
            {
                ["Cold"] = new JsonObject

```

```

        [
            ["High"] = 20,
            ["Low"] = -10
        }
    },
    ["SummaryWords"] = new JsonArray("Cool", "Windy", "Humid")
};

// Add an object.
forecastObject!["TemperatureRanges"]!["Hot"] =
    new JsonObject { ["High"] = 60, ["Low"] = 20 };

// Remove a property.
forecastObject.Remove("SummaryWords");

// Change the value of a property.
forecastObject["Date"] = new DateTime(2019, 8, 3);

var options = new JsonSerializerOptions { WriteIndented = true };
Console.WriteLine(forecastObject.ToString(options));
//output:
//{
//    "Date": "2019-08-03T00:00:00",
//    "Temperature": 25,
//    "Summary": "Hot",
//    "DatesAvailable": [
//        "2019-08-01T00:00:00",
//        "2019-08-02T00:00:00"
//    ],
//    "TemperatureRanges": {
//        "Cold": {
//            "High": 20,
//            "Low": -10
//        },
//        "Hot": {
//            "High": 60,
//            "Low": 20
//        }
//    }
//}
}
}

```

Deserialize subsections of a JSON payload

The following example shows how to use [JsonNode](#) to navigate to a subsection of a JSON tree and deserialize a single value, a custom type, or an array from that subsection.

```
using System.Text.Json;
using System.Text.Json.Nodes;

namespace JsonNodePOCOExample;

public class TemperatureRanges : Dictionary<string, HighLowTemps>
{
}

public class HighLowTemps
{
    public int High { get; set; }
    public int Low { get; set; }
}

public class Program
{
    public static DateTime[]? DatesAvailable { get; set; }

    public static void Main()
    {
        string jsonString =
@"{
    ""Date"": ""2019-08-01T00:00:00"",
    ""Temperature"": 25,
    ""Summary"": ""Hot"",
    ""DatesAvailable"": [
        ""2019-08-01T00:00:00"",
        ""2019-08-02T00:00:00""
    ],
    ""TemperatureRanges"": {
        ""Cold"": {
            ""High"": 20,
            ""Low"": -10
        },
        ""Hot"": {
            ""High"": 60,
            ""Low"": 20
        }
    }
};";
        // Parse all of the JSON.
        JsonNode forecastNode = JsonNode.Parse(jsonString)!;

        // Get a single value
        int hotHigh = forecastNode["TemperatureRanges"]!["Hot"]!
            ["High"]!.GetValue<int>();
        Console.WriteLine($"Hot.High={hotHigh}");
        // output:
        //Hot.High=60

        // Get a subsection and deserialize it into a custom type.
        JsonObject temperatureRangesObject = forecastNode!
```

```

["TemperatureRanges"]!.AsObject();
    using var stream = new MemoryStream();
    using var writer = new Utf8JsonWriter(stream);
    temperatureRangesObject.WriteTo(writer);
    writer.Flush();
    TemperatureRanges? temperatureRanges =
        JsonSerializer.Deserialize<TemperatureRanges>(stream.ToArray());
    Console.WriteLine($"Cold.Low={temperatureRanges!["Cold"].Low},
Hot.High={temperatureRanges["Hot"].High}");
    // output:
    //Cold.Low=-10, Hot.High=60

    // Get a subsection and deserialize it into an array.
    JSONArray datesAvailable = forecastNode!
    ["DatesAvailable"]!.AsArray()!;
    Console.WriteLine($"DatesAvailable[0]={datesAvailable[0]}");
    // output:
    //DatesAvailable[0]=8/1/2019 12:00:00 AM
}
}

```

JsonNode average grade example

The following example selects a JSON array that has integer values and calculates an average value:

C#

```

using System.Text.Json.Nodes;

namespace JsonNodeAverageGradeExample;

public class Program
{
    public static void Main()
    {
        string jsonString =
@"
{
    ""Class Name"": ""Science"",
    ""Teacher\u0027s Name"": ""Jane"",
    ""Semester"": ""2019-01-01"",
    ""Students"": [
        {
            ""Name"": ""John"",
            ""Grade"": 94.3
        },
        {
            ""Name"": ""James"",
            ""Grade"": 81.0
        },
        {
            ""Name"": ""Julia"",

```

```

        """Grade"": 91.9
    },
    {
        """Name"": """",
        """Grade"": 72.4
    },
    {
        """Name"": """Johnathan"""
    }
],
"""Final"": true
}
";
    double sum = 0;
    int count = 0;

    JsonNode document = JsonNode.Parse(jsonString)!;

    JsonNode root = document.Root;
    JsonArray studentsArray = root["Students"]!.AsArray();

    count = studentsArray.Count;

    foreach (JsonNode? student in studentsArray)
    {
        if (student?["Grade"] is JsonNode gradeNode)
        {
            sum += (double)gradeNode;
        }
        else
        {
            sum += 70;
        }
    }

    double average = sum / count;
    Console.WriteLine( $"Average grade : {average}");
}
}
// output:
//Average grade : 81.92

```

The preceding code:

- Calculates an average grade for objects in a `Students` array that have a `Grade` property.
- Assigns a default grade of 70 for students who don't have a grade.
- Gets the number of students from the `Count` property of `JsonArray`.

`JsonNode` with `JsonSerializerOptions`

You can use `JsonSerializer` to serialize and deserialize an instance of `JsonNode`. However, if you use an overload that takes `JsonSerializerOptions`, the options instance is only used to get custom converters. Other features of the options instance are not used. For example, if you set `JsonSerializerOptions.DefaultIgnoreCondition` to `WhenWritingNull` and call `JsonSerializer` with an overload that takes `JsonSerializerOptions`, null properties won't be ignored.

The same limitation applies to the `JsonNode` methods that take a `JsonSerializerOptions` parameter: `WriteTo(Utf8JsonWriter, JsonSerializerOptions)` and `ToJsonObject(JsonSerializerOptions)`. These APIs use `JsonSerializerOptions` only to get custom converters.

The following example illustrates the result of using methods that take a `JsonSerializerOptions` parameter and serialize a `JsonNode` instance:

C#

```
using System.Text;
using System.Text.Json;
using System.Text.Json.Nodes;
using System.Text.Json.Serialization;

namespace JsonNodeWithJsonSerializerOptions;

public class Program
{
    public static void Main()
    {
        Person person = new Person { Name = "Nancy" };

        // Default serialization - Address property included with null
        // token.
        // Output: {"Name":"Nancy","Address":null}
        string personJsonWithNull = JsonSerializer.Serialize(person);
        Console.WriteLine(personJsonWithNull);

        // Serialize and ignore null properties - null Address property is
        // omitted
        // Output: {"Name":"Nancy"}
        JsonSerializerOptions options = new()
        {
            DefaultIgnoreCondition = JsonIgnoreCondition.WhenWritingNull
        };
        string personJsonWithoutNull = JsonSerializer.Serialize(person,
options);
        Console.WriteLine(personJsonWithoutNull);

        // Ignore null properties doesn't work when serializing JsonNode
        // instance
        // by using JsonSerializer.
```

```

        // Output: {"Name":"Nancy", "Address":null}
        var personJsonNode = JsonSerializer.Deserialize<JsonNode>
(personJsonWithNull);
        personJsonWithNull = JsonSerializer.Serialize(personJsonNode,
options);
        Console.WriteLine(personJsonWithNull);

        // Ignore null properties doesn't work when serializing JsonNode
instance
        // by using JsonNode.ToString method.
        // Output: {"Name":"Nancy", "Address":null}
        personJsonWithNull = personJsonNode!.ToString(options);
        Console.WriteLine(personJsonWithNull);

        // Ignore null properties doesn't work when serializing JsonNode
instance
        // by using JsonNode.WriteTo method.
        // Output: {"Name":"Nancy", "Address":null}
        using var stream = new MemoryStream();
        using var writer = new Utf8JsonWriter(stream);
        personJsonNode!.WriteTo(writer, options);
        writer.Flush();
        personJsonWithNull = Encoding.UTF8.GetString(stream.ToArray());
        Console.WriteLine(personJsonWithNull);
    }
}

public class Person
{
    public string? Name { get; set; }
    public string? Address { get; set; }
}

```

If you need features of `JsonSerializerOptions` other than custom converters, use `JsonSerializer` with strongly typed targets (such as the `Person` class in this example) rather than `JsonNode`.

Use `JsonDocument`

The following example shows how to use the `JsonDocument` class for random access to data in a JSON string:

C#

```

double sum = 0;
int count = 0;

using (JsonDocument document = JsonDocument.Parse(jsonString))
{
    JsonElement root = document.RootElement;
    JsonElement studentsElement = root.GetProperty("Students");

```

```

foreach (JsonElement student in studentsElement.EnumerateArray())
{
    if (student.TryGetProperty("Grade", out JsonElement gradeElement))
    {
        sum += gradeElement.GetDouble();
    }
    else
    {
        sum += 70;
    }
    count++;
}

double average = sum / count;
Console.WriteLine($"Average grade : {average}");

```

The preceding code:

- Assumes the JSON to analyze is in a string named `jsonString`.
- Calculates an average grade for objects in a `students` array that have a `Grade` property.
- Assigns a default grade of 70 for students who don't have a grade.
- Creates the `JsonDocument` instance in a `using statement` because `JsonDocument` implements `IDisposable`. After a `JsonDocument` instance is disposed, you lose access to all of its `JsonElement` instances also. To retain access to a `JsonElement` instance, make a copy of it before the parent `JsonDocument` instance is disposed. To make a copy, call `JsonElement.Clone`. For more information, see [JsonDocument is IDisposable](#).

The preceding example code counts students by incrementing a `count` variable with each iteration. An alternative is to call `GetArrayLength`, as shown in the following example:

C#

```

double sum = 0;
int count = 0;

using (JsonDocument document = JsonDocument.Parse(jsonString))
{
    JsonElement root = document.RootElement;
    JsonElement studentsElement = root.GetProperty("Students");

    count = studentsElement.GetArrayLength();

    foreach (JsonElement student in studentsElement.EnumerateArray())
    {

```

```
        if (student.TryGetProperty("Grade", out JsonElement gradeElement))
    {
        sum += gradeElement.GetDouble();
    }
    else
    {
        sum += 70;
    }
}

double average = sum / count;
Console.WriteLine($"Average grade : {average}");
```

Here's an example of the JSON that this code processes:

JSON

```
{
    "Class Name": "Science",
    "Teacher\u0027s Name": "Jane",
    "Semester": "2019-01-01",
    "Students": [
        {
            "Name": "John",
            "Grade": 94.3
        },
        {
            "Name": "James",
            "Grade": 81.0
        },
        {
            "Name": "Julia",
            "Grade": 91.9
        },
        {
            "Name": "Jessica",
            "Grade": 72.4
        },
        {
            "Name": "Johnathan"
        }
    ],
    "Final": true
}
```

For a similar example that uses `JsonNode` instead of `JsonDocument`, see [JsonNode average grade example](#).

How to search a `JsonDocument` and `JsonElement` for sub-elements

Searches on `JsonElement` require a sequential search of the properties and hence are relatively slow (for example when using `TryGetProperty`). `System.Text.Json` is designed to minimize initial parse time rather than lookup time. Therefore, use the following approaches to optimize performance when searching through a `JsonDocument` object:

- Use the built-in enumerators (`EnumerateArray` and `EnumerateObject`) rather than doing your own indexing or loops.
- Don't do a sequential search on the whole `JsonDocument` through every property by using `RootElement`. Instead, search on nested JSON objects based on the known structure of the JSON data. For example, the preceding code examples look for a `Grade` property in `Student` objects by looping through the `Student` objects and getting the value of `Grade` for each, rather than searching through all `JsonElement` objects looking for `Grade` properties. Doing the latter would result in unnecessary passes over the same data.

Use `JsonDocument` to write JSON

The following example shows how to write JSON from a `JsonDocument`:

C#

```
string jsonString = File.ReadAllText(inputFileName);

var writerOptions = new JsonWriterOptions
{
    Indented = true
};

var documentOptions = new JsonDocumentOptions
{
    CommentHandling = JsonCommentHandling.Skip
};

using FileStream fs = File.Create(outputFileName);
using var writer = new Utf8JsonWriter(fs, options: writerOptions);
using JsonDocument document = JsonDocument.Parse(jsonString,
documentOptions);

JsonElement root = document.RootElement;

if (root.ValueKind == JsonValueKind.Object)
{
    writer.WriteStartObject();
```

```

    }

    else
    {
        return;
    }

    foreach (JsonProperty property in root.EnumerateObject())
    {
        property.WriteTo(writer);
    }

    writer.WriteEndObject();

    writer.Flush();

```

The preceding code:

- Reads a JSON file, loads the data into a `JsonDocument`, and writes formatted (pretty-printed) JSON to a file.
- Uses `JsonDocumentOptions` to specify that comments in the input JSON are allowed but ignored.
- When finished, calls `Flush` on the writer. An alternative is to let the writer auto-flush when it's disposed.

Here's an example of JSON input to be processed by the example code:

JSON

```
{"Class Name": "Science", "Teacher's Name": "Jane", "Semester": "2019-01-01", "Students": [{"Name": "John", "Grade": 94.3}, {"Name": "James", "Grade": 81.0}, {"Name": "Julia", "Grade": 91.9}, {"Name": "Jessica", "Grade": 72.4}, {"Name": "Johnathan"}], "Final": true}
```

The result is the following pretty-printed JSON output:

JSON

```
{
    "Class Name": "Science",
    "Teacher\u0027s Name": "Jane",
    "Semester": "2019-01-01",
    "Students": [
        {
            "Name": "John",
            "Grade": 94.3
        },
        {
            "Name": "James",
            "Grade": 81.0
        }
    ]
}
```

```
        },
        {
            "Name": "Julia",
            "Grade": 91.9
        },
        {
            "Name": "Jessica",
            "Grade": 72.4
        },
        {
            "Name": "Johnathan"
        }
    ],
    "Final": true
}
```

JsonDocument is IDisposable

`JsonDocument` builds an in-memory view of the data into a pooled buffer. Therefore the `JsonDocument` type implements `IDisposable` and needs to be used inside a `using` block.

Only return a `JsonDocument` from your API if you want to transfer lifetime ownership and dispose responsibility to the caller. In most scenarios, that isn't necessary. If the caller needs to work with the entire JSON document, return the `Clone` of the `RootElement`, which is a `JsonElement`. If the caller needs to work with a particular element within the JSON document, return the `Clone` of that `JsonElement`. If you return the `RootElement` or a sub-element directly without making a `clone`, the caller won't be able to access the returned `JsonElement` after the `JsonDocument` that owns it is disposed.

Here's an example that requires you to make a `Clone`:

C#

```
public JsonElement LookAndFeel(JsonElement source)
{
    string json =
File.ReadAllText(source.GetProperty("fileName").GetString());

    using (JsonDocument doc = JsonDocument.Parse(json))
    {
        return doc.RootElement.Clone();
    }
}
```

The preceding code expects a `JsonElement` that contains a `fileName` property. It opens the JSON file and creates a `JsonDocument`. The method assumes that the caller wants to

work with the entire document, so it returns the `Clone` of the `RootElement`.

If you receive a `JsonElement` and are returning a sub-element, it's not necessary to return a `Clone` of the sub-element. The caller is responsible for keeping alive the `JsonDocument` that the passed-in `JsonElement` belongs to. For example:

C#

```
public JsonElement ReturnFileName(JsonElement source)
{
    return source.GetProperty("fileName");
}
```

JsonDocument with JsonSerializerOptions

You can use `JsonSerializer` to serialize and deserialize an instance of `JsonDocument`. However, the implementation for reading and writing `JsonDocument` instances by using `JsonSerializer` is a wrapper over the `JsonDocument.ParseValue(Utf8JsonReader)` and `JsonDocument.WriteTo(Utf8JsonWriter)`. This wrapper does not forward any `JsonSerializerOptions` (serializer features) to `Utf8JsonReader` or `Utf8JsonWriter`. For example, if you set `JsonSerializerOptions.DefaultIgnoreCondition` to `WhenWritingNull` and call `JsonSerializer` with an overload that takes `JsonSerializerOptions`, null properties won't be ignored.

The following example illustrates the result of using methods that take a `JsonSerializerOptions` parameter and serialize a `JsonDocument` instance:

C#

```
using System.Text;
using System.Text.Json;
using System.Text.Json.Nodes;
using System.Text.Json.Serialization;

namespace JsonDocumentWithJsonSerializerOptions;

public class Program
{
    public static void Main()
    {
        Person person = new Person { Name = "Nancy" };

        // Default serialization - Address property included with null token.
        // Output: {"Name":"Nancy","Address":null}
        string personJsonWithNull = JsonSerializer.Serialize(person);
```

```

        Console.WriteLine(personJsonWithNull);

        // Serialize and ignore null properties - null Address property is
omitted
        // Output: {"Name":"Nancy"}
        JsonSerializerOptions options = new()
        {
            DefaultIgnoreCondition = JsonIgnoreCondition.WhenWritingNull
        };
        string personJsonWithoutNull = JsonSerializer.Serialize(person,
options);
        Console.WriteLine(personJsonWithoutNull);

        // Ignore null properties doesn't work when serializing JsonDocument
instance
        // by using JsonSerializer.
        // Output: {"Name":"Nancy","Address":null}
        var personJsonDocument = JsonSerializer.Deserialize<JsonDocument>
(personJsonWithNull);
        personJsonWithNull = JsonSerializer.Serialize(personJsonDocument,
options);
        Console.WriteLine(personJsonWithNull);
    }
}

public class Person
{
    public string? Name { get; set; }

    public string? Address { get; set; }
}

```

If you need features of `JsonSerializerOptions`, use `JsonSerializer` with strongly typed targets (such as the `Person` class in this example) rather than `JsonDocument`.

See also

- [System.Text.Json overview](#)
- [System.Text.Json API reference](#)
- [System.Text.Json.Serialization API reference](#)

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

more information, see [our contributor guide](#).

 [Provide product feedback](#)

How to use Utf8JsonWriter in System.Text.Json

Article • 05/26/2023

This article shows how to use the [Utf8JsonWriter](#) type for building custom serializers.

[Utf8JsonWriter](#) is a high-performance way to write UTF-8 encoded JSON text from common .NET types like `String`, `Int32`, and `DateTime`. The writer is a low-level type that can be used to build custom serializers. The [JsonSerializer.Serialize](#) method uses `Utf8JsonWriter` under the covers.

The following example shows how to use the [Utf8JsonWriter](#) class:

C#

```
var options = new JsonWriterOptions
{
    Indented = true
};

using var stream = new MemoryStream();
using var writer = new Utf8JsonWriter(stream, options);

writer.WriteStartObject();
writer.WriteString("date", DateTimeOffset.UtcNow);
writer.WriteNumber("temp", 42);
writer.WriteEndObject();
writer.Flush();

string json = Encoding.UTF8.GetString(stream.ToArray());
Console.WriteLine(json);
```

Write with UTF-8 text

To achieve the best possible performance while using the `Utf8JsonWriter`, write JSON payloads already encoded as UTF-8 text rather than as UTF-16 strings. Use [JsonEncodedText](#) to cache and pre-encode known string property names and values as statics, and pass those to the writer, rather than using UTF-16 string literals. This is faster than caching and using UTF-8 byte arrays.

This approach also works if you need to do custom escaping. `System.Text.Json` doesn't let you disable escaping while writing a string. However, you could pass in your own custom [JavaScriptEncoder](#) as an option to the writer, or create your own

`JsonEncodedText` that uses your `JavascriptEncoder` to do the escaping, and then write the `JsonEncodedText` instead of the string. For more information, see [Customize character encoding](#).

Write raw JSON

In some scenarios, you might want to write "raw" JSON to a JSON payload that you're creating with `Utf8JsonWriter`. You can use `Utf8JsonWriter.WriteRawValue` to do that. Here are typical scenarios:

- You have an existing JSON payload that you want to enclose in new JSON.
- You want to format values differently from the default `Utf8JsonWriter` formatting.

For example, you might want to customize number formatting. By default, `System.Text.Json` omits the decimal point for whole numbers, writing `1` rather than `1.0`, for example. The rationale is that writing fewer bytes is good for performance. But suppose the consumer of your JSON treats numbers with decimals as doubles, and numbers without decimals as integers. You might want to ensure that the numbers in an array are all recognized as doubles, by writing a decimal point and zero for whole numbers. The following example shows how to do that:

C#

```
using System.Text;
using System.Text.Json;

namespace WriteRawJson;

public class Program
{
    public static void Main()
    {
        JsonWriterOptions writerOptions = new() { Indented = true, };

        using MemoryStream stream = new();
        using Utf8JsonWriter writer = new(stream, writerOptions);

        writer.WriteStartObject();

        writer.WriteStartArray("defaultJsonFormatting");
        foreach (double number in new double[] { 50.4, 51 })
        {
            writer.WriteStartObject();
            writer.WritePropertyName("value");
            writer.WriteNumberValue(number);
            writer.WriteEndObject();
        }
    }
}
```

```

        }

        writer.WriteEndArray();

        writer.WriteStartArray("customJsonFormatting");
        foreach (double result in new double[] { 50.4, 51 })
        {
            writer.WriteStartObject();
            writer.WritePropertyName("value");
            writer.WriteRawValue(
                FormatNumberValue(result), skipInputValidation: true);
            writer.WriteEndObject();
        }
        writer.WriteEndArray();

        writer.WriteEndObject();
        writer.Flush();

        string json = Encoding.UTF8.GetString(stream.ToArray());
        Console.WriteLine(json);
    }
    static string FormatNumberValue(double numberValue)
    {
        return numberValue == Convert.ToInt32(numberValue) ?
            numberValue.ToString() + ".0" : numberValue.ToString();
    }
}

// output:
//{
//  "defaultJsonFormatting": [
//    {
//      "value": 50.4
//    },
//    {
//      "value": 51
//    }
//  ],
//  "customJsonFormatting": [
//    {
//      "value": 50.4
//    },
//    {
//      "value": 51.0
//    }
//  ]
//}

```

Customize character escaping

The [StringEscapeHandling](#) setting of `JsonTextWriter` offers options to escape all non-ASCII characters or HTML characters. By default, `Utf8JsonWriter` escapes all non-ASCII and HTML characters. This escaping is done for defense-in-depth security reasons. To

specify a different escaping policy, create a `JavaScriptEncoder` and set `JsonWriterOptions.Encoder`. For more information, see [Customize character encoding](#).

Write null values

To write null values by using `Utf8JsonWriter`, call:

- `WriteNull` to write a key-value pair with null as the value.
- `WriteNullValue` to write null as an element of a JSON array.

For a string property, if the string is null, `WriteString` and `WriteStringValue` are equivalent to `WriteNull` and `WriteNullValue`.

Write Timespan, Uri, or char values

To write `Timespan`, `Uri`, or `char` values, format them as strings (by calling `ToString()`, for example) and call `WriteStringValue`.

See also

- [How to use Utf8JsonReader in System.Text.Json](#)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

How to use Utf8JsonReader in System.Text.Json

Article • 06/21/2023

This article shows how you can use the [Utf8JsonReader](#) type for building custom parsers and deserializers.

[Utf8JsonReader](#) is a high-performance, low allocation, forward-only reader for UTF-8 encoded JSON text, read from a `ReadOnlySpan<byte>` or `ReadOnlySequence<byte>`. The [Utf8JsonReader](#) is a low-level type that can be used to build custom parsers and deserializers. The [JsonSerializer.Deserialize](#) methods use [Utf8JsonReader](#) under the covers.

`Utf8JsonReader` can't be used directly from Visual Basic code. For more information, see [Visual Basic support](#).

The following example shows how to use the [Utf8JsonReader](#) class:

C#

```
var options = new JsonReaderOptions
{
    AllowTrailingCommas = true,
    CommentHandling = JsonCommentHandling.Skip
};
var reader = new Utf8JsonReader(jsonUtf8Bytes, options);

while (reader.Read())
{
    Console.Write(reader.TokenType);

    switch (reader.TokenType)
    {
        case JsonTokenType.PropertyName:
        case JsonTokenType.String:
            {
                string? text = reader.GetString();
                Console.Write(" ");
                Console.Write(text);
                break;
            }

        case JsonTokenType.Number:
            {
                int intValue = reader.GetInt32();
                Console.Write(" ");
            }
    }
}
```

```

        Console.WriteLine(intValue);
        break;
    }

    // Other token types elided for brevity
}
Console.WriteLine();
}

```

The preceding code assumes that the `jsonUtf8` variable is a byte array that contains valid JSON, encoded as UTF-8.

Filter data using `Utf8JsonReader`

The following example shows how to synchronously read a file and search for a value.

C#

```

using System.Text;
using System.Text.Json;

namespace SystemTextJsonSamples
{
    public class Utf8ReaderFromFile
    {
        private static readonly byte[] s_nameUtf8 =
Encoding.UTF8.GetBytes("name");
        private static ReadOnlySpan<byte> Utf8Bom => new byte[] { 0xEF,
0xBB, 0xBF };

        public static void Run()
        {
            // ReadAllBytes if the file encoding is UTF-8:
            string fileName = "UniversitiesUtf8.json";
            ReadOnlySpan<byte> jsonReadOnlySpan =
File.ReadAllBytes(fileName);

            // Read past the UTF-8 BOM bytes if a BOM exists.
            if (jsonReadOnlySpan.StartsWith(Utf8Bom))
            {
                jsonReadOnlySpan = jsonReadOnlySpan.Slice(Utf8Bom.Length);
            }

            // Or read as UTF-16 and transcode to UTF-8 to convert to a
ReadOnlySpan<byte>
            //string fileName = "Universities.json";
            //string jsonString = File.ReadAllText(fileName);
            //ReadOnlySpan<byte> jsonReadOnlySpan =
Encoding.UTF8.GetBytes(jsonString);

            int count = 0;

```

```

int total = 0;

var reader = new Utf8JsonReader(jsonReadOnlySpan);

while (reader.Read())
{
    JsonTokenType tokenType = reader.TokenType;

    switch (tokenType)
    {
        case JsonTokenType.StartObject:
            total++;
            break;
        case JsonTokenType.PropertyName:
            if (reader.ValueTextEquals(s_nameUtf8))
            {
                // Assume valid JSON, known schema
                reader.Read();
                if (reader.GetString()!.EndsWith("University"))
                {
                    count++;
                }
            }
            break;
    }
}

Console.WriteLine($"{count} out of {total} have names that end
with 'University'");
}
}
}

```

For an asynchronous version of this example, see [.NET samples JSON project](#).

The preceding code:

- Assumes the JSON contains an array of objects and each object might contain a "name" property of type string.
- Counts objects and "name" property values that end with "University".
- Assumes the file is encoded as UTF-16 and transcodes it into UTF-8. A file encoded as UTF-8 can be read directly into a `ReadOnlySpan<byte>` by using the following code:

C#

```
ReadOnlySpan<byte> jsonReadOnlySpan = File.ReadAllBytes(fileName);
```

If the file contains a UTF-8 byte order mark (BOM), remove it before passing the bytes to the `Utf8JsonReader`, since the reader expects text. Otherwise, the BOM is considered invalid JSON, and the reader throws an exception.

Here's a JSON sample that the preceding code can read. The resulting summary message is "2 out of 4 have names that end with 'University'" :

```
JSON
[  
  {  
    "web_pages": [ "https://contoso.edu/" ],  
    "alpha_two_code": "US",  
    "state-province": null,  
    "country": "United States",  
    "domains": [ "contoso.edu" ],  
    "name": "Contoso Community College"  
  },  
  {  
    "web_pages": [ "http://fabrikam.edu/" ],  
    "alpha_two_code": "US",  
    "state-province": null,  
    "country": "United States",  
    "domains": [ "fabrikam.edu" ],  
    "name": "Fabrikam Community College"  
  },  
  {  
    "web_pages": [ "http://www.contosouniversity.edu/" ],  
    "alpha_two_code": "US",  
    "state-province": null,  
    "country": "United States",  
    "domains": [ "contosouniversity.edu" ],  
    "name": "Contoso University"  
  },  
  {  
    "web_pages": [ "http://www.fabrikamuniversity.edu/" ],  
    "alpha_two_code": "US",  
    "state-province": null,  
    "country": "United States",  
    "domains": [ "fabrikamuniversity.edu" ],  
    "name": "Fabrikam University"  
  }  
]
```

Read from a stream using `Utf8JsonReader`

When reading a large file (a gigabyte or more in size, for example), you might want to avoid having to load the entire file into memory at once. For this scenario, you can use a `FileStream`.

When using the `Utf8JsonReader` to read from a stream, the following rules apply:

- The buffer containing the partial JSON payload must be at least as large as the largest JSON token within it so that the reader can make forward progress.
- The buffer must be at least as large as the largest sequence of white space within the JSON.
- The reader doesn't keep track of the data it has read until it completely reads the next `TokenType` in the JSON payload. So when there are bytes left over in the buffer, you have to pass them to the reader again. You can use `BytesConsumed` to determine how many bytes are left over.

The following code illustrates how to read from a stream. The example shows a `MemoryStream`. Similar code will work with a `FileStream`, except when the `FileStream` contains a UTF-8 BOM at the start. In that case, you need to strip those three bytes from the buffer before passing the remaining bytes to the `Utf8JsonReader`. Otherwise the reader would throw an exception, since the BOM is not considered a valid part of the JSON.

The sample code starts with a 4 KB buffer and doubles the buffer size each time it finds that the size is not large enough to fit a complete JSON token, which is required for the reader to make forward progress on the JSON payload. The JSON sample provided in the snippet triggers a buffer size increase only if you set a very small initial buffer size, for example, 10 bytes. If you set the initial buffer size to 10, the `Console.WriteLine` statements illustrate the cause and effect of buffer size increases. At the 4 KB initial buffer size, the entire sample JSON is shown by each `Console.WriteLine`, and the buffer size never has to be increased.

C#

```
using System.Text;
using System.Text.Json;

namespace SystemTextJsonSamples
{
    public class Utf8ReaderPartialRead
    {
        public static void Run()
        {
            var jsonString = @"
                ""Date"": ""2019-08-01T00:00:00-07:00"",
                ""Temperature"": 25,
                ""TemperatureRanges"": {
                    ""Cold"": { ""High"": 20, ""Low"": -10 },
                    ""Hot"": { ""High"": 60, ""Low"": 20 }
                },
                ""Summary"": ""Hot"",
            ";
            // ...
        }
    }
}
```

```
    }";

    byte[] bytes = Encoding.UTF8.GetBytes(jsonString);
    var stream = new MemoryStream(bytes);

    var buffer = new byte[4096];

    // Fill the buffer.
    // For this snippet, we're assuming the stream is open and has
data.
    // If it might be closed or empty, check if the return value is
0.
    stream.Read(buffer);

    // We set isFinalBlock to false since we expect more data in a
subsequent read from the stream.
    var reader = new Utf8JsonReader(buffer, isFinalBlock: false,
state: default);
    Console.WriteLine($"String in buffer is:
{Encoding.UTF8.GetString(buffer)}");

    // Search for "Summary" property name
    while (reader.TokenType != JsonTokenType.PropertyName ||
!reader.ValueTextEquals("Summary"))
    {
        if (!reader.Read())
        {
            // Not enough of the JSON is in the buffer to complete a
read.
            GetMoreBytesFromStream(stream, ref buffer, ref reader);
        }
    }

    // Found the "Summary" property name.
    Console.WriteLine($"String in buffer is:
{Encoding.UTF8.GetString(buffer)}");
    while (!reader.Read())
    {
        // Not enough of the JSON is in the buffer to complete a
read.
        GetMoreBytesFromStream(stream, ref buffer, ref reader);
    }
    // Display value of Summary property, that is, "Hot".
    Console.WriteLine($"Got property value: {reader.GetString()}");
}

private static void GetMoreBytesFromStream(
    MemoryStream stream, ref byte[] buffer, ref Utf8JsonReader
reader)
{
    int bytesRead;
    if (reader.BytesConsumed < buffer.Length)
    {
        ReadOnlySpan<byte> leftover =
buffer.AsSpan((int)reader.BytesConsumed);
```

```

        if (leftover.Length == buffer.Length)
        {
            Array.Resize(ref buffer, buffer.Length * 2);
            Console.WriteLine($"Increased buffer size to
{buffer.Length}");
        }

        leftover.CopyTo(buffer);
        bytesRead = stream.Read(buffer.AsSpan(leftover.Length));
    }
    else
    {
        bytesRead = stream.Read(buffer);
    }
    Console.WriteLine($"String in buffer is:
{Encoding.UTF8.GetString(buffer)}");
    reader = new Utf8JsonReader(buffer, isFinalBlock: bytesRead ==
0, reader.CurrentState);
}
}
}

```

The preceding example sets no limit to how large the buffer can grow. If the token size is too large, the code could fail with an [OutOfMemoryException](#) exception. This can happen if the JSON contains a token that is around 1 GB or more in size, because doubling the 1 GB size results in a size that is too large to fit into an `int32` buffer.

ref struct limitations

Because the `Utf8JsonReader` type is a *ref struct*, it has [certain limitations](#). For example, it can't be stored as a field on a class or struct other than a ref struct.

To achieve high performance, this type must be a `ref struct` since it needs to cache the input `ReadOnlySpan<byte>`, which itself is a ref struct. In addition, the `Utf8JsonReader` type is mutable since it holds state. Therefore, **pass it by reference** rather than by value. Passing it by value would result in a struct copy and the state changes would not be visible to the caller.

For more information about how to use ref structs, see [Avoid allocations](#).

Read UTF-8 text

To achieve the best possible performance while using `Utf8JsonReader`, read JSON payloads already encoded as UTF-8 text rather than as UTF-16 strings. For a code example, see [Filter data using Utf8JsonReader](#).

Read with multi-segment `ReadOnlySequence`

If your JSON input is a `ReadOnlySpan<byte>`, each JSON element can be accessed from the `ValueSpan` property on the reader as you go through the read loop. However, if your input is a `ReadOnlySequence<byte>` (which is the result of reading from a `PipeReader`), some JSON elements might straddle multiple segments of the `ReadOnlySequence<byte>` object. These elements would not be accessible from `ValueSpan` in a contiguous memory block. Instead, whenever you have a multi-segment `ReadOnlySequence<byte>` as input, poll the `HasValueSequence` property on the reader to figure out how to access the current JSON element. Here's a recommended pattern:

C#

```
while (reader.Read())
{
    switch (reader.TokenType)
    {
        // ...
        ReadOnlySpan<byte> jsonElement = reader.HasValueSequence ?
            reader.ValueSequence.ToArray() :
            reader.ValueSpan;
        // ...
    }
}
```

Use `ValueTextEquals` for property name lookups

Don't use `ValueSpan` to do byte-by-byte comparisons by calling `SequenceEqual` for property name lookups. Call `ValueTextEquals` instead, because that method unescapes any characters that are escaped in the JSON. Here's an example that shows how to search for a property that's named "name":

C#

```
private static readonly byte[] s_nameUtf8 = Encoding.UTF8.GetBytes("name");
```

C#

```
while (reader.Read())
{
    switch (reader.TokenType)
    {
        case JsonTokenType.StartObject:
```

```
        total++;
        break;
    case JsonTokenType.PropertyName:
        if (reader.ValueTextEquals(s_nameUtf8))
        {
            count++;
        }
        break;
    }
}
```

Read null values into nullable value types

The built-in `System.Text.Json` APIs return only non-nullable value types. For example, `Utf8JsonReader.GetBoolean` returns a `bool`. It throws an exception if it finds `Null` in the JSON. The following examples show two ways to handle nulls, one by returning a nullable value type and one by returning the default value:

C#

```
public bool? ReadAsNullableBoolean()
{
    _reader.Read();
    if (_reader.TokenType == JsonTokenType.Null)
    {
        return null;
    }
    if (_reader.TokenType != JsonTokenType.True && _reader.TokenType != JsonTokenType.False)
    {
        throw new JsonException();
    }
    return _reader.GetBoolean();
}
```

C#

```
public bool ReadAsBoolean(bool defaultValue)
{
    _reader.Read();
    if (_reader.TokenType == JsonTokenType.Null)
    {
        return defaultValue;
    }
    if (_reader.TokenType != JsonTokenType.True && _reader.TokenType != JsonTokenType.False)
    {
        throw new JsonException();
    }
}
```

```
        return _reader.GetBoolean();
    }
```

Skip children of token

Use the [Utf8JsonReader.Skip\(\)](#) method to skip the children of the current JSON token. If the token type is [JsonTokenType.PropertyName](#), the reader moves to the property value. The following code snippet shows an example of using [Utf8JsonReader.Skip\(\)](#) to move the reader to the value of a property.

C#

```
var weatherForecast = new WeatherForecast
{
    Date = DateTime.Parse("2019-08-01"),
    TemperatureCelsius = 25,
    Summary = "Hot"
};

byte[] jsonUtf8Bytes = JsonSerializer.SerializeToUtf8Bytes(weatherForecast);

var reader = new Utf8JsonReader(jsonUtf8Bytes);

int temp;
while (reader.Read())
{
    switch (reader.TokenType)
    {
        case JsonTokenType.PropertyName:
            {
                if (reader.ValueTextEquals("TemperatureCelsius"))
                {
                    reader.Skip();
                    temp = reader.GetInt32();

                    Console.WriteLine($"Temperature is {temp} degrees.");
                }
                continue;
            }
        default:
            continue;
    }
}
```

Consume decoded JSON strings

Starting in .NET 7, you can use the [Utf8JsonReader.CopyString](#) method instead of [Utf8JsonReader.GetString\(\)](#) to consume a decoded JSON string. Unlike [GetString\(\)](#), which always allocates a new string, [CopyString](#) lets you copy the unescaped string to a buffer that you own. The following code snippet shows an example of consuming a UTF-16 string using [CopyString](#).

C#

```
var reader = new Utf8JsonReader( /* jsonReadOnlySpan */ );  
  
int valueLength = reader.HasValueSequence  
    ? checked((int)reader.ValueSequence.Length)  
    : reader.ValueSpan.Length;  
  
char[] buffer = ArrayPool<char>.Shared.Rent(valueLength);  
int charsRead = reader.CopyString(buffer);  
ReadOnlySpan<char> source = buffer.AsSpan(0, charsRead);  
  
// Handle the unescaped JSON string.  
ParseUnescapedString(source);  
ArrayPool<char>.Shared.Return(buffer, clearArray: true);  
  
void ParseUnescapedString(ReadOnlySpan<char> source)  
{  
    // ...  
}
```

Related APIs

- To deserialize a custom type from a `Utf8JsonReader` instance, call `JsonSerializer.Deserialize< TValue >(Utf8JsonReader, JsonSerializerOptions)` or `JsonSerializer.Deserialize< TValue >(Utf8JsonReader, JsonTypeInfo< TValue >)`. For an example, see [Deserialize from UTF-8](#).
- `JsonNode` and the classes that derive from it provide the ability to create a mutable DOM. You can convert a `Utf8JsonReader` instance to a `JsonNode` by calling `JsonNode.Parse(Utf8JsonReader, Nullable<JsonNodeOptions>)`. The following code snippet shows an example.

C#

```
using System.Text.Json;  
using System.Text.Json.Nodes;  
  
namespace Utf8ReaderToJsonNode  
{  
    public class WeatherForecast
```

```

    }

    public DateTimeOffset Date { get; set; }
    public int TemperatureCelsius { get; set; }
    public string? Summary { get; set; }
}

public class Program
{
    public static void Main()
    {
        var weatherForecast = new WeatherForecast
        {
            Date = DateTime.Parse("2019-08-01"),
            TemperatureCelsius = 25,
            Summary = "Hot"
        };

        byte[] jsonUtf8Bytes =
JsonSerializer.SerializeToUtf8Bytes(weatherForecast);

        var utf8Reader = new Utf8JsonReader(jsonUtf8Bytes);
        JsonNode? node = JsonNode.Parse(ref utf8Reader);
        Console.WriteLine(node);
    }
}

```

- [JsonDocument](#) provides the ability to build a read-only DOM by using `Utf8JsonReader`. Call the `JsonDocument.ParseValue(Utf8JsonReader)` method to parse a `JsonDocument` from a `Utf8JsonReader` instance. You can access the JSON elements that compose the payload via the `JsonElement` type. For example code that uses `JsonDocument.ParseValue(Utf8JsonReader)`, see [RoundtripDataTable.cs](#) and the code snippet in [Deserialize inferred types to object properties](#).
- You can also parse a `Utf8JsonReader` instance to a `JsonElement`, which represents a specific JSON value, by calling `JsonElement.ParseValue(Utf8JsonReader)`.

See also

- [How to use Utf8JsonWriter in System.Text.Json](#)

 Collaborate with us on
GitHub

The source for this content can
be found on GitHub, where you



.NET feedback

The .NET documentation is open
source. Provide feedback [here](#).

can also create and review issues and pull requests. For more information, see [our contributor guide](#).

 Open a documentation issue

 Provide product feedback

Visual Basic support

Article • 09/23/2022

Parts of System.Text.Json use [ref structs](#), which are not supported by Visual Basic. If you try to use System.Text.Json ref struct APIs with Visual Basic you get BC40000 compiler errors. The error message indicates that the problem is an obsolete API, but the actual issue is lack of ref struct support in the compiler. The following parts of System.Text.Json aren't usable from Visual Basic:

- The [Utf8JsonReader](#) class. Since the [JsonConverter<T>.Read](#) method takes a `Utf8JsonReader` parameter, this limitation means you can't use Visual Basic to write custom converters. A workaround for this is to implement custom converters in a C# library assembly, and reference that assembly from your VB project. This assumes that all you do in Visual Basic is register the converters into the serializer. You can't call the `Read` methods of the converters from Visual Basic code.
- Overloads of other APIs that include a [ReadOnlySpan<T>](#) type. Most methods include overloads that use `String` instead of `ReadOnlySpan`.

These restrictions are in place because ref structs cannot be used safely without language support, even when just "passing data through." Subverting this error will result in Visual Basic code that can corrupt memory and should not be done.

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

Supported collection types in System.Text.Json

Article • 12/05/2023

This article gives an overview of which collections are supported for serialization and deserialization. [System.Text.Json.JsonSerializer](#) supports a collection type for serialization if it:

- Derives from [IEnumerable](#) or [IAsyncEnumerable<T>](#)
- Contains elements that are serializable.

The serializer calls the [GetEnumerator\(\)](#) method and writes the elements.

Deserialization is more complicated and is not supported for some collection types.

The following sections are organized by namespace and show which types are supported for serialization and deserialization.

System.Array namespace

[] Expand table

| Type | Serialization | Deserialization |
|---------------------------|---------------|-----------------|
| Single-dimensional arrays | ✓ | ✓ |
| Multi-dimensional arrays | ✗ | ✗ |
| Jagged arrays | ✓ | ✓ |

System.Collections namespace

[] Expand table

| Type | Serialization | Deserialization |
|-----------------|---------------|-----------------|
| ArrayList | ✓ | ✓ |
| BitArray | ✓ | ✗ |
| DictionaryEntry | ✓ | ✓ |

| Type | Serialization | Deserialization |
|-------------|---------------|-----------------|
| Hashtable | ✓ | ✓ |
| ICollection | ✓ | ✓ |
| IDictionary | ✓ | ✓ |
| IEnumerable | ✓ | ✓ |
| IList | ✓ | ✓ |
| Queue | ✓ | ✓ |
| SortedList | ✓ | ✓ |
| Stack * | ✓ | ✓ |

* See [Support round trip for Stack types.](#)

System.Collections.Generic namespace

[\[+\] Expand table](#)

| Type | Serialization | Deserialization |
|---------------------------------------|---------------|-----------------|
| Dictionary< TKey, TValue > * | ✓ | ✓ |
| HashSet<T> | ✓ | ✓ |
| IAsyncEnumerable<T> † | ✓ | ✓ |
| ICollection<T> | ✓ | ✓ |
| IDictionary< TKey, TValue > * | ✓ | ✓ |
| IEnumerable<T> | ✓ | ✓ |
| IList<T> | ✓ | ✓ |
| IReadOnlyCollection<T> | ✓ | ✓ |
| IReadOnlyDictionary< TKey, TValue > * | ✓ | ✓ |
| IReadOnlyList<T> | ✓ | ✓ |
| ISet<T> | ✓ | ✓ |
| KeyValuePair< TKey, TValue > | ✓ | ✓ |

| Type | Serialization | Deserialization |
|------------------------------------|---------------|-----------------|
| LinkedList<T> | ✓ | ✓ |
| LinkedListNode<T> | ✓ | ✗ |
| List<T> | ✓ | ✓ |
| Queue<T> | ✓ | ✓ |
| SortedDictionary< TKey, TValue > * | ✓ | ✓ |
| SortedList< TKey, TValue > * | ✓ | ✓ |
| SortedSet<T> | ✓ | ✓ |
| Stack<T> ‡ | ✓ | ✓ |

* See [Supported key types](#).

† See the following section on [IAsyncEnumerable<T>](#).

‡ See [Support round trip for Stack types](#).

IAsyncEnumerable<T>

The following examples use streams as a representation of any async source of data. The source could be files on a local machine, or results from a database query or web service API call.

Stream serialization

`System.Text.Json` supports serializing [IAsyncEnumerable<T>](#) values as JSON arrays, as shown in the following example:

```
C#  
  
using System.Text.Json;  
  
namespace IAsyncEnumerableSerialize;  
  
public class Program  
{  
    public static async Task Main()  
    {  
        using Stream stream = Console.OpenStandardOutput();  
        var data = new { Data = PrintNumbers(3) };  
        await JsonSerializer.SerializeAsync(stream, data);  
    }  
}  
  
private static string PrintNumbers(int count)  
{  
    var numbers = new List<int>();  
    for (int i = 0; i < count; i++)  
    {  
        numbers.Add(i);  
    }  
    return string.Join(", ", numbers);  
}
```

```
}

    static async IAsyncEnumerable<int> PrintNumbers(int n)
    {
        for (int i = 0; i < n; i++)
        {
            await Task.Delay(1000);
            yield return i;
        }
    }
// output:
// {"Data": [0,1,2]}
```

`IAsyncEnumerable<T>` values are only supported by the asynchronous serialization methods, such as [JsonSerializer.SerializeAsync](#).

Stream deserialization

The `DeserializeAsyncEnumerable` method supports streaming deserialization, as shown in the following example:

C#

```
using System.Text;
using System.Text.Json;

namespace IAsyncEnumerableDeserialize;

public class Program
{
    public static async Task Main()
    {
        using var stream = new MemoryStream(Encoding.UTF8.GetBytes("[0,1,2,3,4]"));
        await foreach (int item in
JsonSerializer.DeserializeAsyncEnumerable<int>(stream))
        {
            Console.WriteLine(item);
        }
    }
// output:
//0
//1
//2
//3
//4
```

The `DeserializeAsyncEnumerable` method only supports reading from root-level JSON arrays.

The `DeserializeAsync` method supports `IAsyncEnumerable<T>`, but its signature doesn't allow streaming. It returns the final result as a single value, as shown in the following example.

C#

```
using System.Text;
using System.Text.Json;

namespace IAsyncEnumerableDeserializeNonStreaming;

public class MyPoco
{
    public IAsyncEnumerable<int>? Data { get; set; }
}

public class Program
{
    public static async Task Main()
    {
        using var stream = new MemoryStream(Encoding.UTF8.GetBytes(@"
{""Data""::[0,1,2,3,4]}"));
        MyPoco? result = await JsonSerializer.DeserializeAsync<MyPoco>
(stream)!;
        await foreach (int item in result!.Data!)
        {
            Console.WriteLine(item);
        }
    }
}
// output:
//0
//1
//2
//3
//4
```

In this example, the deserializer buffers all `IAsyncEnumerable<T>` contents in memory before returning the deserialized object. This behavior is necessary because the deserializer needs to read the entire JSON payload before returning a result.

System.Collections.Immutable namespace

[] Expand table

| Type | Serialization | Deserialization |
|--|---------------|-----------------|
| IImmutableDictionary<TKey,TValue> † | ✓ | ✓ |
| IImmutableList<T> | ✓ | ✓ |
| IImmutableQueue<T> | ✓ | ✓ |
| IImmutableSet<T> | ✓ | ✓ |
| IImmutableStack<T> * | ✓ | ✓ |
| ImmutableArray<T> | ✓ | ✓ |
| ImmutableDictionary<TKey,TValue> † | ✓ | ✓ |
| ImmutableHashSet<T> | ✓ | ✓ |
| ImmutableQueue<T> | ✓ | ✓ |
| ImmutableSortedDictionary<TKey,TValue> † | ✓ | ✓ |
| ImmutableSortedSet<T> | ✓ | ✓ |
| ImmutableStack<T> * | ✓ | ✓ |

* See [Support round trip for Stack types](#).

† See [Supported key types](#).

System.Collections.Specialized namespace

[] Expand table

| Type | Serialization | Deserialization |
|---------------------|---------------|-----------------|
| BitVector32 | ✓ | ✗ * |
| HybridDictionary | ✓ | ✓ |
| IOrderedDictionary | ✓ | ✗ |
| ListDictionary | ✓ | ✓ |
| NameValueCollection | ✓ | ✗ |
| StringCollection | ✓ | ✗ |
| StringDictionary | ✓ | ✗ |

* When `BitVector32` is deserialized, the `Data` property is skipped because it doesn't have a public setter. No exception is thrown.

System.Collections.Concurrent namespace

[+] Expand table

| Type | Serialization | Deserialization |
|--|---------------|-----------------|
| <code>BlockingCollection<T></code> | ✓ | ✗ |
| <code>ConcurrentBag<T></code> | ✓ | ✗ |
| <code>ConcurrentDictionary<TKey,TValue></code> † | ✓ | ✓ |
| <code>ConcurrentQueue<T></code> | ✓ | ✓ |
| <code>ConcurrentStack<T></code> * | ✓ | ✓ |

* See [Support round trip for Stack types](#).

† See [Supported key types](#).

System.Collections.ObjectModel namespace

[+] Expand table

| Type | Serialization | Deserialization |
|--|---------------|-----------------|
| <code>Collection<T></code> | ✓ | ✓ |
| <code>KeyedCollection<string, TValue></code> * | ✓ | ✗ |
| <code>ObservableCollection<T></code> | ✓ | ✓ |
| <code>ReadOnlyCollection<T></code> | ✓ | ✗ |
| <code>ReadOnlyDictionary<TKey,TValue></code> | ✓ | ✗ |
| <code>ReadOnlyObservableCollection<T></code> | ✓ | ✗ |

* Non-`string` keys are not supported.

Custom collections

Any collection type that isn't in one of the preceding namespaces is considered a custom collection. Such types include user-defined types and types defined by ASP.NET Core. For example, [Microsoft.Extensions.Primitives](#) is in this group.

All custom collections (everything that derives from `IEnumerable`) are supported for serialization, as long as their element types are supported.

Custom collections with deserialization support

A custom collection is supported for deserialization if it:

- Isn't an interface or abstract.
- Has a parameterless constructor.
- Contains element types that are supported by [JsonSerializer](#).
- Implements or inherits one or more of the following interfaces or classes:
 - [ConcurrentQueue<T>](#)
 - [ConcurrentStack<T>](#) *
 - [ICollection<T>](#)
 - [IDictionary](#)
 - [IDictionary<TKey,TValue>](#) †
 - [IList](#)
 - [IList<T>](#)
 - [Queue](#)
 - [Queue<T>](#)
 - [Stack](#) *
 - [Stack<T>](#) *

* See [Support round trip for Stack types](#).

† See [Supported key types](#).

Custom collections with known issues

There are known issues with the following custom collections:

- [ExpandoObject](#): See [dotnet/runtime#29690](#) ↴.
- [DynamicObject](#): See [dotnet/runtime#1808](#) ↴.
- [DataTable](#): See [dotnet/docs#21366](#) ↴.
- [Microsoft.AspNetCore.Http.FormFile](#): See [dotnet/runtime#1559](#) ↴.
- [Microsoft.AspNetCore.Http.IFormCollection](#): See [dotnet/runtime#1559](#) ↴.

For more information about known issues, see the [open issues in System.Text.Json](#).

Supported key types

Supported types for the keys of `Dictionary` and `SortedList` types include the following:

- `Boolean`
- `Byte`
- `DateTime`
- `DateTimeOffset`
- `Decimal`
- `Double`
- `Enum`
- `Guid`
- `Int16`
- `Int32`
- `Int64`
- `Object` (Only on serialization and if the runtime type is one of the supported types in this list.)
- `SByte`
- `Single`
- `String`
- `UInt16`
- `UInt32`
- `UInt64`

System.Data namespace

There are no built-in converters for `DataSet`, `DataTable`, and related types in the `System.Data` namespace. Deserializing these types from untrusted input is not safe, as explained in [the security guidance](#). However, you can write a custom converter to support these types. For sample custom converter code that serializes and deserializes a `DataTable`, see [RoundtripDataTable.cs](#).

See also

- [Populate initialized properties](#)
- [System.Text.Json overview](#)

- [System.Text.Json API reference](#)
- [System.Text.Json.Serialization API reference](#)

 **Collaborate with us on GitHub**

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

.NET is an open source project.
Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

Reflection versus source generation in System.Text.Json

Article • 11/01/2023

This article explains the differences between reflection and source generation as it relates to `System.Text.Json` serialization. It also provides guidance on how to choose the best approach for your scenario.

Metadata collection

To serialize or deserialize a type, `JsonSerializer` needs information about how to access the members of the type. `JsonSerializer` needs the following information:

- How to access property getters and fields for serialization.
- How to access a constructor, property setters, and fields for deserialization.
- Information about which attributes have been used to customize serialization or deserialization.
- Run-time configuration from `JsonSerializerOptions`.

This information is referred to as *metadata*.

Reflection

By default, `JsonSerializer` collects metadata at run time by using `reflection`. Whenever `JsonSerializer` has to serialize or deserialize a type for the first time, it collects and caches this metadata. The metadata collection process takes time and uses memory.

Source generation

As an alternative, `System.Text.Json` can use the C# `source generation` feature to improve performance, reduce private memory usage, and facilitate `assembly trimming`, which reduces app size. In addition, certain reflection APIs can't be used in `Native AOT applications`, so you must use source generation for those apps.

Source generation can be used in two modes:

- **Metadata-based mode**

During compilation, `System.Text.Json` collects the information needed for serialization and generates source code files that populate JSON contract metadata for the requested types.

- **Serialization-optimization (fast path) mode**

`JsonSerializer` features that customize the output of serialization, such as naming policies and reference preservation, carry a performance overhead. In serialization-optimization mode, `System.Text.Json` generates optimized serialization code that uses `Utf8JsonWriter` directly. This optimized or *fast path* code increases serialization throughput.

Fast-path *deserialization* isn't currently available. For more information, see [dotnet/runtime issue 55043 ↗](#).

Source generation for `System.Text.Json` requires C# 9.0 or a later version.

Feature comparison

Choose reflection or source-generation modes based on the following benefits that each one offers:

| Benefit | Reflection | Source generation (Metadata-based mode) | Source generation (Serialization-optimization mode) |
|--|------------|--|--|
| Simpler to code. | ✓ | ✗ | ✗ |
| Simpler to debug. | ✗ | ✓ | ✓ |
| Supports non-public members. | ✓ | ✓ [*] | ✓ [*] |
| Supports all available serialization customizations. | ✓ | ✗ [†] | ✗ [†] |
| Reduces start-up time. | ✗ | ✓ | ✓ |
| Reduces private memory usage. | ✗ | ✓ | ✓ |
| Eliminates run-time reflection. | ✗ | ✓ | ✓ |
| Facilitates trim-safe app size reduction. | ✗ | ✓ | ✓ |
| Increases serialization throughput. | ✗ | ✗ | ✓ |

* The source generator supports *some* non-public members, for example, internal types in the same assembly. † Source-generated contracts can be modified using the contract customization API.

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

Source-generation modes in System.Text.Json

Article • 11/01/2023

Source generation can be used in two modes: metadata-based and serialization optimization. This article describes the different modes.

For information about how to use source generation modes, see [How to use source generation in System.Text.Json](#).

Metadata-based mode

You can use source generation to move the metadata collection process from run time to compile time. During compilation, the metadata is collected and source code files are generated. The generated source code files are automatically compiled as an integral part of the application. This technique eliminates run-time metadata collection, which improves performance of both serialization and deserialization.

The performance improvements provided by source generation can be substantial. For example, [test results](#) have shown up to 40% or more startup time reduction, private memory reduction, throughput speed increase (in serialization optimization mode), and app size reduction.

Known issues

Only `public` properties and fields are supported by default in either serialization mode. However, reflection mode supports the use of `private` members and `private accessors`, while source-generation mode doesn't. For example, if you apply the `JsonInclude attribute` to a `private` property or a property that has a `private` setter or getter, it will be serialized in reflection mode. Source-generation mode supports only `public` or `internal` members and `public` or `internal` accessors of `public` properties. If you set `[JsonInclude]` on `private` members or accessors and choose source-generation mode, a `NotSupportedException` will be thrown at run time.

For information about other known issues with source generation, see the [GitHub issues that are labeled "source-generator"](#) in the `dotnet/runtime` repository.

Serialization-optimization (fast path) mode

`JsonSerializer` has many features that customize the output of serialization, such as [naming policies](#) and [preserving references](#). Support for all those features causes some performance overhead. Source generation can improve serialization performance by generating optimized code that uses [Utf8JsonWriter](#) directly.

The optimized code doesn't support all of the serialization features that `JsonSerializer` supports. The serializer detects whether the optimized code can be used and falls back to default serialization code if unsupported options are specified. For example, [JsonNumberHandling.AllowReadingFromString](#) isn't applicable to writing, so specifying this option doesn't cause a fallback to default code.

The following table shows which options in `JsonSerializerOptions` are supported by fast-path serialization:

| Serialization option | Supported for fast-path |
|---------------------------------------|-------------------------|
| <code>AllowTrailingCommas</code> | ✓ |
| <code>Converters</code> | ✗ |
| <code>DefaultBufferSize</code> | ✓ |
| <code>DefaultIgnoreCondition</code> | ✓ |
| <code>DictionaryKeyPolicy</code> | ✗ |
| <code>Encoder</code> | ✗ |
| <code>IgnoreNullValues</code> | ✗ |
| <code>IgnoreReadOnlyFields</code> | ✓ |
| <code>IgnoreReadOnlyProperties</code> | ✓ |
| <code>IncludeFields</code> | ✓ |
| <code>MaxDepth</code> | ✓ |
| <code>NumberHandling</code> | ✗ |
| <code>PropertyNamingPolicy</code> | ✓ |
| <code>ReferenceHandler</code> | ✗ |
| <code>TypeInfoResolver</code> | ✓ |
| <code>WriteIndented</code> | ✓ |

(The following options aren't supported because they apply only to *deserialization*: [PropertyNameCaseInsensitive](#), [ReadCommentHandling](#), and [UnknownTypeHandling](#).)

The following table shows which attributes are supported by fast-path serialization:

| Attribute | Supported for fast-path |
|---|-------------------------|
| JsonConstructorAttribute | ✗ |
| JsonConverterAttribute | ✗ |
| JsonDerivedTypeAttribute | ✓ |
| JsonExtensionDataAttribute | ✗ |
| JsonIgnoreAttribute | ✓ |
| JsonIncludeAttribute | ✓ |
| JsonNumberHandlingAttribute | ✗ |
| JsonPolymorphicAttribute | ✓ |
| JsonPropertyNameAttribute | ✓ |
| JsonPropertyOrderAttribute | ✗ |
| JsonRequiredAttribute | ✓ |

If a non-supported option or attribute is specified for a type, the serializer falls back to metadata mode, assuming that the source generator has been configured to generate metadata. In that case, the optimized code isn't used when serializing that type but may be used for other types. Therefore it's important to do performance testing with your options and workloads to determine how much benefit you can actually get from serialization-optimization mode. Also, the ability to fall back to `JsonSerializer` code requires metadata-collection mode. If you select only serialization-optimization mode, serialization might fail for types or options that need to fall back to `JsonSerializer` code.

See also

- [JSON serialization and deserialization in .NET - overview](#)
- [How to use the library](#)

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

How to use source generation in System.Text.Json

Article • 12/16/2023

Source generation in System.Text.Json is available in .NET 6 and later versions. When used in an app, the app's language version must be C# 9.0 or later. This article shows you how to use source-generation-backed serialization in your apps.

For information about the different source-generation modes, see [Source-generation modes](#).

Use source-generation defaults

To use source generation with all defaults (both modes, default options):

1. Create a partial class that derives from [JsonSerializerContext](#).
2. Specify the type to serialize or deserialize by applying [JsonSerializableAttribute](#) to the context class.
3. Call a [JsonSerializer](#) method that either:
 - Takes a [JsonTypeInfo<T>](#) instance, or
 - Takes a [JsonSerializerContext](#) instance, or
 - Takes a [JsonSerializerOptions](#) instance and you've set its [JsonSerializerOptions.TypeInfoResolver](#) property to the `Default` property of the context type (.NET 7 and later only).

By default, both source generation modes are used if you don't specify one. For information about how to specify the mode to use, see [Specify source generation mode](#) later in this article.

Here's the type that is used in the following examples:

C#

```
public class WeatherForecast
{
    public DateTime Date { get; set; }
    public int TemperatureCelsius { get; set; }
    public string? Summary { get; set; }
}
```

Here's the context class configured to do source generation for the preceding `WeatherForecast` class:

C#

```
[JsonSourceGenerationOptions(WriteIndented = true)]
[JsonSerializable(typeof(WeatherForecast))]
internal partial class SourceGenerationContext : JsonSerializerContext
{}
```

The types of `WeatherForecast` members don't need to be explicitly specified with `[JsonSerializable]` attributes. Members declared as `object` are an exception to this rule. The runtime type for a member declared as `object` needs to be specified. For example, suppose you have the following class:

C#

```
public class WeatherForecast
{
    public object? Data { get; set; }
    public List<object>? DataList { get; set; }
}
```

And you know that at runtime it may have `boolean` and `int` objects:

C#

```
WeatherForecast wf = new() { Data = true, DataList = new List<object> {
    true, 1 } };
```

Then `boolean` and `int` have to be declared as `[JsonSerializable]`:

C#

```
[JsonSerializable(typeof(WeatherForecast))]
[JsonSerializable(typeof(bool))]
[JsonSerializable(typeof(int))]
public partial class WeatherForecastContext : JsonSerializerContext
{}
```

To specify source generation for a collection, use `[JsonSerializable]` with the collection type. For example: `[JsonSerializable(typeof(List<WeatherForecast>))]`.

JsonSerializer methods that use source generation

In the following examples, the static `Default` property of the context type provides an instance of the context type with default options. The context instance provides a `WeatherForecast` property that returns a `JsonTypeInfo<WeatherForecast>` instance. You can specify a different name for this property by using the `TypeInfoPropertyName` property of the `[JsonSerializable]` attribute.

Serialization examples

Using `JsonTypeInfo<T>`:

C#

```
jsonString = JsonSerializer.Serialize(
    weatherForecast!, SourceGenerationContext.Default.WeatherForecast);
```

Using `JsonSerializerContext`:

C#

```
jsonString = JsonSerializer.Serialize(
    weatherForecast, typeof(WeatherForecast),
    SourceGenerationContext.Default);
```

Using `JsonSerializerOptions`:

C#

```
sourceGenOptions = new JsonSerializerOptions
{
    TypeInfoResolver = SourceGenerationContext.Default
};

jsonString = JsonSerializer.Serialize(
    weatherForecast, typeof(WeatherForecast), sourceGenOptions);
```

Deserialization examples

Using `JsonTypeInfo<T>`:

C#

```
weatherForecast = JsonSerializer.Deserialize<WeatherForecast>(
    jsonString, SourceGenerationContext.Default.WeatherForecast);
```

Using [JsonSerializerContext](#):

C#

```
weatherForecast = JsonSerializer.Deserialize(
    jsonString, typeof(WeatherForecast), SourceGenerationContext.Default)
    as WeatherForecast;
```

Using [JsonSerializerOptions](#):

C#

```
var sourceGenOptions = new JsonSerializerOptions
{
    TypeInfoResolver = SourceGenerationContext.Default
};
weatherForecast = JsonSerializer.Deserialize(
    jsonString, typeof(WeatherForecast), sourceGenOptions)
    as WeatherForecast;
```

Complete program example

Here are the preceding examples in a complete program:

C#

```
using System.Text.Json;
using System.Text.Json.Serialization;

namespace BothModesNoOptions
{
    public class WeatherForecast
    {
        public DateTime Date { get; set; }
        public int TemperatureCelsius { get; set; }
        public string? Summary { get; set; }
    }

    [JsonSourceGenerationOptions(WriteIndented = true)]
    [JsonSerializable(typeof(WeatherForecast))]
    internal partial class SourceGenerationContext : JsonSerializerContext
    {

    }

    public class Program
```

```
{
    public static void Main()
    {
        string jsonString =
@"{
    ""Date"": ""2019-08-01T00:00:00"",
    ""TemperatureCelsius"": 25,
    ""Summary"": ""Hot"""
}
";
        WeatherForecast? weatherForecast;

        weatherForecast = JsonSerializer.Deserialize<WeatherForecast>(
            jsonString,
SourceGenerationContext.Default.WeatherForecast);
        Console.WriteLine($"Date={weatherForecast?.Date}");
        // output:
        //Date=8/1/2019 12:00:00 AM

        weatherForecast = JsonSerializer.Deserialize(
            jsonString, typeof(WeatherForecast),
SourceGenerationContext.Default)
        as WeatherForecast;
        Console.WriteLine($"Date={weatherForecast?.Date}");
        // output:
        //Date=8/1/2019 12:00:00 AM

        var sourceGenOptions = new JsonSerializerOptions
{
    TypeInfoResolver = SourceGenerationContext.Default
};
        weatherForecast = JsonSerializer.Deserialize(
            jsonString, typeof(WeatherForecast), sourceGenOptions)
        as WeatherForecast;
        Console.WriteLine($"Date={weatherForecast?.Date}");
        // output:
        //Date=8/1/2019 12:00:00 AM

        jsonString = JsonSerializer.Serialize(
            weatherForecast!,
SourceGenerationContext.Default.WeatherForecast);
        Console.WriteLine(jsonString);
        // output:
        //{"Date":"2019-08-
01T00:00:00","TemperatureCelsius":25,"Summary":"Hot"}

        jsonString = JsonSerializer.Serialize(
            weatherForecast, typeof(WeatherForecast),
SourceGenerationContext.Default);
        Console.WriteLine(jsonString);
        // output:
        //{"Date":"2019-08-
01T00:00:00","TemperatureCelsius":25,"Summary":"Hot"}

        sourceGenOptions = new JsonSerializerOptions
```

```

    {
        TypeInfoResolver = SourceGenerationContext.Default
    };

    jsonString = JsonSerializer.Serialize(
        weatherForecast, typeof(WeatherForecast), sourceGenOptions);
    Console.WriteLine(jsonString);
    // output:
    // {"Date":"2019-08-
01T00:00:00", "TemperatureCelsius":25, "Summary":"Hot"}
}
}
}

```

Specify source-generation mode

You can specify metadata-based mode or serialization-optimization mode for an entire context, which may include multiple types. Or you can specify the mode for an individual type. If you do both, the mode specification for a type wins.

- For an entire context, use the [JsonSourceGenerationOptionsAttribute.GenerationMode](#) property.
- For an individual type, use the [JsonSerializableAttribute.GenerationMode](#) property.

Serialization-optimization (fast path) mode example

- For an entire context:

```
C#
[JsonSourceGenerationOptions\(GenerationMode =
JsonSourceGenerationMode.Serialization\)]
[JsonSerializable\(typeof\(WeatherForecast\)\)]
internal partial class SerializeOnlyContext : JsonSerializerContext
{
}
```

- For an individual type:

```
C#
[JsonSerializable\(typeof\(WeatherForecast\), GenerationMode =
JsonSourceGenerationMode.Serialization\)]
internal partial class SerializeOnlyWeatherForecastOnlyContext : JsonSerializerContext
```

```
{  
}
```

- Complete program example

```
C#
```

```
using System.Text.Json;  
using System.Text.Json.Serialization;  
  
namespace SerializeOnlyNoOptions  
{  
    public class WeatherForecast  
    {  
        public DateTime Date { get; set; }  
        public int TemperatureCelsius { get; set; }  
        public string? Summary { get; set; }  
    }  
  
    [JsonSourceGenerationOptions(GenerationMode =  
JsonSourceGenerationMode.Serialization)]  
    [JsonSerializable(typeof(WeatherForecast))]  
    internal partial class SerializeOnlyContext : JsonSerializerContext  
    {  
    }  
  
    [JsonSerializable(typeof(WeatherForecast), GenerationMode =  
JsonSourceGenerationMode.Serialization)]  
    internal partial class SerializeOnlyWeatherForecastOnlyContext :  
JsonSerializerContext  
    {  
    }  
  
    public class Program  
    {  
        public static void Main()  
        {  
            string jsonString;  
            WeatherForecast weatherForecast = new()  
            { Date = DateTime.Parse("2019-08-01"),  
TemperatureCelsius = 25, Summary = "Hot" };  
  
            // Use context that selects Serialization mode only for  
WeatherForecast.  
            jsonString = JsonSerializer.Serialize(weatherForecast,  
SerializeOnlyWeatherForecastOnlyContext.Default.WeatherForecast);  
            Console.WriteLine(jsonString);  
            // output:  
            //{"Date":"2019-08-  
01T00:00:00","TemperatureCelsius":25,"Summary":"Hot"}  
  
            // Use a context that selects Serialization mode.
```

```
        jsonString = JsonSerializer.Serialize(weatherForecast,
            SerializeOnlyContext.Default.WeatherForecast);
        Console.WriteLine(jsonString);
        // output:
        // {"Date":"2019-08-
01T00:00:00","TemperatureCelsius":25,"Summary":"Hot"}
    }
}
```

Metadata-based mode example

- For an entire context:

C#

```
[JsonSourceGenerationOptions(GenerationMode =
JsonSourceGenerationMode.Metadata)]
[JsonSerializable(typeof(WeatherForecast))]
internal partial class MetadataOnlyContext : JsonSerializerContext
{}
```

C#

```
jsonString = JsonSerializer.Serialize(
    weatherForecast!, MetadataOnlyContext.Default.WeatherForecast);
```

C#

```
weatherForecast = JsonSerializer.Deserialize<WeatherForecast>(
    jsonString, MetadataOnlyContext.Default.WeatherForecast);
```

- For an individual type:

C#

```
[JsonSerializable(typeof(WeatherForecast), GenerationMode =
JsonSourceGenerationMode.Metadata)]
internal partial class MetadataOnlyWeatherForecastOnlyContext :
JsonSerializerContext
{}
```

C#

```
jsonString = JsonSerializer.Serialize(
    weatherForecast!,
    MetadataOnlyWeatherForecastOnlyContext.Default.WeatherForecast);
```

C#

```
weatherForecast = JsonSerializer.Deserialize<WeatherForecast>(
    jsonString,
    MetadataOnlyWeatherForecastOnlyContext.Default.WeatherForecast);
```

- Complete program example

C#

```
using System.Text.Json;
using System.Text.Json.Serialization;

namespace MetadataOnlyNoOptions
{
    public class WeatherForecast
    {
        public DateTime Date { get; set; }
        public int TemperatureCelsius { get; set; }
        public string? Summary { get; set; }
    }

    [JsonSerializable(typeof(WeatherForecast), GenerationMode =
JsonSourceGenerationMode.Metadata)]
    internal partial class MetadataOnlyWeatherForecastOnlyContext : JsonSerializerContext
    {
    }

    [JsonSourceGenerationOptions(GenerationMode =
JsonSourceGenerationMode.Metadata)]
    [JsonSerializable(typeof(WeatherForecast))]
    internal partial class MetadataOnlyContext : JsonSerializerContext
    {
    }

    public class Program
    {
        public static void Main()
        {
            string jsonString =
@"
{
    ""Date"": ""2019-08-01T00:00:00"",
    ""TemperatureCelsius"": 25,
    ""Summary"": ""Hot""
}
";
```

```

        WeatherForecast? weatherForecast;

        // Deserialize with context that selects metadata mode only
        // for WeatherForecast only.
        weatherForecast =
JsonSerializer.Deserialize<WeatherForecast>(
            jsonString,
MetadataOnlyWeatherForecastOnlyContext.Default.WeatherForecast);
            Console.WriteLine($"Date={weatherForecast?.Date}");
            // output:
            //Date=8/1/2019 12:00:00 AM

        // Serialize with context that selects metadata mode only
        // for WeatherForecast only.
        jsonString = JsonSerializer.Serialize(
            weatherForecast!,

MetadataOnlyWeatherForecastOnlyContext.Default.WeatherForecast);
            Console.WriteLine(jsonString);
            // output:
            //{"Date":"2019-08-
01T00:00:00","TemperatureCelsius":25,"Summary":"Hot"}

        // Deserialize with context that selects metadata mode
        // only.
        weatherForecast =
JsonSerializer.Deserialize<WeatherForecast>(
            jsonString,
MetadataOnlyContext.Default.WeatherForecast);
            Console.WriteLine($"Date={weatherForecast?.Date}");
            // output:
            //Date=8/1/2019 12:00:00 AM

        // Serialize with context that selects metadata mode only.
        jsonString = JsonSerializer.Serialize(
            weatherForecast!,
MetadataOnlyContext.Default.WeatherForecast);
            Console.WriteLine(jsonString);
            // output:
            //{"Date":"2019-08-
01T00:00:00","TemperatureCelsius":0,"Summary":"Hot"}
        }
    }
}

```

Source-generation support in ASP.NET Core

In Blazor apps, use overloads of [HttpClientJsonExtensions.GetFromJsonAsync](#) and [HttpClientJsonExtensions.PostAsJsonAsync](#) extension methods that take a source generation context or `TypeInfo< TValue >`.

Starting with .NET 8, you can also use overloads of `HttpClientJsonExtensions.GetFromJsonAsAsyncEnumerable` extension methods that accept a source generation context or `TypeInfo< TValue >`.

In Razor Pages, MVC, SignalR, and Web API apps, use the `JsonSerializerOptions.TypeInfoResolver` property to specify the context.

C#

```
[JsonSerializable(typeof(WeatherForecast[]))]
internal partial class MyJsonContext : JsonSerializerContext { }
```

C#

```
var serializerOptions = new JsonSerializerOptions
{
    TypeInfoResolver = MyJsonContext.Default;
};

services.AddControllers().AddJsonOptions(
    static options =>

    options.JsonSerializerOptions.TypeInfoResolverChain.Add(MyJsonContext.Default));

```

ⓘ Note

`JsonSourceGenerationMode.Serialization`, or fast-path serialization, isn't supported for asynchronous serialization.

In .NET 7 and earlier versions, this limitation also applies to synchronous overloads of `JsonSerializer.Serialize` that accept a `Stream`. Starting with .NET 8, even though streaming serialization requires metadata-based models, it will fall back to fast-path if the payloads are known to be small enough to fit in the predetermined buffer size. For more information, see <https://devblogs.microsoft.com/dotnet/performance-improvements-in-net-8/#json>.

Disable reflection defaults

Because `System.Text.Json` uses reflection by default, calling a basic serialization method can break Native AOT apps, which doesn't support all required reflection APIs. These breaks can be challenging to diagnose since they can be unpredictable, and apps are

often debugged using the CoreCLR runtime, where reflection works. Instead, if you explicitly disable reflection-based serialization, breaks are easier to diagnose. Code that uses reflection-based serialization will cause an [InvalidOperationException](#) with a descriptive message to be thrown at run time.

To disable default reflection in your app, set the

`JsonSerializerIsReflectionEnabledByDefault` MSBuild property to `false` in your project file:

XML

```
<PropertyGroup>

<JsonSerializerIsReflectionEnabledByDefault>false</JsonSerializerIsReflectionEnabledByDefault>
</PropertyGroup>
```

- The behavior of this property is consistent regardless of runtime, either CoreCLR or Native AOT.
- If you don't specify this property and [PublishTrimmed](#) is enabled, reflection-based serialization is automatically disabled.

You can programmatically check whether reflection is disabled by using the `JsonSerializer.IsReflectionEnabledByDefault` property. The following code snippet shows how you might configure your serializer depending on whether reflection is enabled:

C#

```
static JsonSerializerOptions CreateDefaultOptions()
{
    return new()
    {
        TypeInfoResolver = JsonSerializer.IsReflectionEnabledByDefault
            ? new DefaultJsonTypeInfoResolver()
            : MyContext.Default
    };
}
```

Because the property is treated as a link-time constant, the previous method doesn't root the reflection-based resolver in applications that run in Native AOT.

Specify options

In .NET 8 and later versions, most options that you can set using `JsonSerializerOptions` can also be set using the `JsonSourceGenerationOptionsAttribute` attribute. The

advantage to setting options via the attribute is that the configuration is specified at compile time, which ensures that the generated `MyContext.Default` property is preconfigured with all the relevant options set.

The following code shows how to set options using the `JsonSourceGenerationOptionsAttribute` attribute.

```
C#
```

```
[JsonSourceGenerationOptions(
    WriteIndented = true,
    PropertyNamingPolicy = JsonKnownNamingPolicy.CamelCase,
    GenerationMode = JsonSourceGenerationMode.Serialization)]
[JsonSerializable(typeof(WeatherForecast))]
internal partial class SerializationModeOptionsContext : 
JsonSerializerContext
{
}
```

When using `JsonSourceGenerationOptionsAttribute` to specify serialization options, call one of the following serialization methods:

- A `JsonSerializer.Serialize` method that takes a `TypeInfo<TValue>`. Pass it the `Default.<TypeName>` property of your context class:

```
C#
```

```
jsonString = JsonSerializer.Serialize(
    weatherForecast,
    SerializationModeOptionsContext.Default.WeatherForecast);
```

- A `JsonSerializer.Serialize` method that takes a context. Pass it the `Default` static property of your context class.

```
C#
```

```
jsonString = JsonSerializer.Serialize(
    weatherForecast, typeof(WeatherForecast),
    SerializationModeOptionsContext.Default);
```

If you call a method that lets you pass in your own instance of `Utf8JsonWriter`, the writer's `Indented` setting is honored instead of the `JsonSourceGenerationOptionsAttribute.WriteIndented` option.

If you create and use a context instance by calling the constructor that takes a `JsonSerializerOptions` instance, the supplied instance will be used instead of the options specified by `JsonSourceGenerationOptionsAttribute`.

Here are the preceding examples in a complete program:

C#

```
using System.Text.Json;
using System.Text.Json.Serialization;

namespace SerializeOnlyWithOptions
{
    public class WeatherForecast
    {
        public DateTime Date { get; set; }
        public int TemperatureCelsius { get; set; }
        public string? Summary { get; set; }
    }

    [JsonSourceGenerationOptions(
        WriteIndented = true,
        PropertyNamingPolicy = JsonKnownNamingPolicy.CamelCase,
        GenerationMode = JsonSourceGenerationMode.Serialization)]
    [JsonSerializable(typeof(WeatherForecast))]
    internal partial class SerializationModeOptionsContext : 
JsonSerializerContext
    {
    }

    public class Program
    {
        public static void Main()
        {
            string jsonString;
            WeatherForecast weatherForecast = new()
                { Date = DateTime.Parse("2019-08-01"), TemperatureCelsius =
25, Summary = "Hot" };

            // Serialize using TypeInfo<TValue> provided by the context
            // and options specified by [JsonSourceGenerationOptions].
            jsonString = JsonSerializer.Serialize(
                weatherForecast,
                SerializationModeOptionsContext.Default.WeatherForecast);
            Console.WriteLine(jsonString);
            // output:
            //{
            //   "date": "2019-08-01T00:00:00",
            //   "temperatureCelsius": 0,
            //   "summary": "Hot"
            //}

            // Serialize using Default context
            // and options specified by [JsonSourceGenerationOptions].
        }
    }
}
```

```

        jsonString = JsonSerializer.Serialize(
            weatherForecast, typeof(WeatherForecast),
            SerializationModeOptionsContext.Default);
        Console.WriteLine(jsonString);
        // output:
        //{
        //   "date": "2019-08-01T00:00:00",
        //   "temperatureCelsius": 0,
        //   "summary": "Hot"
        //}
    }
}

```

Combine source generators

You can combine contracts from multiple source-generated contexts inside a single [JsonSerializerOptions](#) instance. Use the [JsonSerializerOptions.TypeInfoResolver](#) property to chain multiple contexts that have been combined by using the [JsonTypeInfoResolver.Combine\(IJsonTypeInfoResolver\[\]\)](#) method.

C#

```

var options = new JsonSerializerOptions
{
    TypeInfoResolver = JsonTypeInfoResolver.Combine(ContextA.Default,
ContextB.Default, ContextC.Default);
};

```

Starting in .NET 8, if you later want to prepend or append another context, you can do so using the [JsonSerializerOptions.TypeInfoResolverChain](#) property. The ordering of the chain is significant: [JsonSerializerOptions](#) queries each of the resolvers in their specified order and returns the first result that's non-null.

C#

```

options.TypeInfoResolverChain.Add(ContextD.Default); // Append to the end of
// the list.
options.TypeInfoResolverChain.Insert(0, ContextE.Default); // Insert at the
// beginning of the list.

```

Any change made to the [TypeInfoResolverChain](#) property is reflected by [TypeInfoResolver](#) and vice versa.

Serialize enum fields as strings

By default, enums are serialized as numbers. To [serialize a specific enum's fields as strings](#) when using source generation, annotate it with the `JsonStringEnumConverter<TEnum>` converter. Or to set a [blanket policy](#) for all enumerations, use the `JsonSourceGenerationOptionsAttribute` attribute.

JsonStringEnumConverter<T> converter

To serialize enum names as strings using source generation, use the `JsonStringEnumConverter<TEnum>` converter. (The non-generic `JsonStringEnumConverter` type is not supported by the Native AOT runtime.)

Annotate the enumeration type with the `JsonStringEnumConverter<TEnum>` converter using the `JsonConverterAttribute` attribute:

C#

```
public class WeatherForecastWithPrecipEnum
{
    public DateTimeOffset Date { get; set; }
    public int TemperatureCelsius { get; set; }
    public Precipitation? Precipitation { get; set; }
}

[JsonConverter(typeof(JsonStringEnumConverter<Precipitation>))]
public enum Precipitation
{
    Drizzle, Rain, Sleet, Hail, Snow
}
```

Create a `JsonSerializerContext` class and annotate it with the `JsonSerializableAttribute` attribute:

C#

```
[JsonSerializable(typeof(WeatherForecastWithPrecipEnum))]
public partial class Context1 : JsonSerializerContext { }
```

The following code serializes the enum names instead of the numeric values:

C#

```
var weatherForecast = new WeatherForecastWithPrecipEnum
{
    Date = DateTime.Parse("2019-08-01"),
    TemperatureCelsius = 25,
    Precipitation = Precipitation.Sleet
};
```

```
var options = new JsonSerializerOptions
{
    WriteIndented = true,
    TypeInfoResolver = Context1.Default,
};
string? jsonString = JsonSerializer.Serialize(weatherForecast, options);
```

The resulting JSON looks like the following example:

JSON

```
{
    "Date": "2019-08-01T00:00:00-07:00",
    "TemperatureCelsius": 25,
    "Precipitation": "Sleet"
}
```

Blanket policy

Instead of using the `JsonStringEnumConverter<TEnum>` type, you can apply a blanket policy to serialize enums as strings by using the `JsonSourceGenerationOptionsAttribute`. Create a `JsonSerializerContext` class and annotate it with the `JsonSerializableAttribute` and `JsonSourceGenerationOptionsAttribute` attributes:

C#

```
[JsonSourceGenerationOptions(UseStringEnumConverter = true)]
[JsonSerializable(typeof(WeatherForecast2WithPrecipEnum))]
public partial class Context2 : JsonSerializerContext { }
```

Notice that the enum doesn't have the `JsonConverterAttribute`:

C#

```
public class WeatherForecast2WithPrecipEnum
{
    public DateTimeOffset Date { get; set; }
    public int TemperatureCelsius { get; set; }
    public Precipitation2? Precipitation { get; set; }
}

public enum Precipitation2
{
    Drizzle, Rain, Sleet, Hail, Snow
}
```

See also

- [JSON serialization and deserialization in .NET - overview](#)
- [How to use the library](#)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

.NET is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

How to customize character encoding with System.Text.Json

Article • 05/26/2023

By default, the serializer escapes all non-ASCII characters. That is, it replaces them with `\uxxxx` where `xxxx` is the Unicode code of the character. For example, if the `Summary` property in the following JSON is set to Cyrillic `жарко`, the `WeatherForecast` object is serialized as shown in this example:

JSON

```
{  
    "Date": "2019-08-01T00:00:00-07:00",  
    "TemperatureCelsius": 25,  
    "Summary": "\u0436\u0430\u0440\u043A\u043E"  
}
```

Serialize language character sets

To serialize the character set(s) of one or more languages without escaping, specify [Unicode range\(s\)](#) when creating an instance of [System.Text.Encodings.Web.JavaScriptEncoder](#), as shown in the following example:

C#

```
using System.Text.Encodings.Web;  
using System.Text.Json;  
using System.Text.Unicode;
```

C#

```
var options1 = new JsonSerializerOptions  
{  
    Encoder = JavaScriptEncoder.Create(UnicodeRanges.BasicLatin,  
    UnicodeRanges.Cyrillic),  
    WriteIndented = true  
};  
jsonString = JsonSerializer.Serialize(weatherForecast, options1);
```

This code doesn't escape Cyrillic or Greek characters. If the `Summary` property is set to Cyrillic `жарко`, the `WeatherForecast` object is serialized as shown in this example:

JSON

```
{  
    "Date": "2019-08-01T00:00:00-07:00",  
    "TemperatureCelsius": 25,  
    "Summary": "жарко"  
}
```

By default, the encoder is initialized with the [BasicLatin](#) range.

To serialize all language sets without escaping, use [UnicodeRanges.All](#).

Serialize specific characters

An alternative is to specify individual characters that you want to allow through without being escaped. The following example serializes only the first two characters of жарко:

C#

```
using System.Text_ENCODINGS.Web;  
using System.Text.Json;  
using System.Text.Unicode;
```

C#

```
var encoderSettings = new TextEncoderSettings();  
encoderSettings.AllowCharacters('\u0436', '\u0430');  
encoderSettings.AllowRange(UnicodeRanges.BasicLatin);  
var options2 = new JsonSerializerOptions  
{  
    Encoder = JavaScriptEncoder.Create(encoderSettings),  
    WriteIndented = true  
};  
jsonString = JsonSerializer.Serialize(weatherForecast, options2);
```

Here's an example of JSON produced by the preceding code:

JSON

```
{  
    "Date": "2019-08-01T00:00:00-07:00",  
    "TemperatureCelsius": 25,  
    "Summary": "\u043A\u0430\u043B\u043E\u0436\u0435\u043D\u0438\u044F"  
}
```

Block lists

The preceding sections show how to specify allow lists of code points or ranges that you don't want to be escaped. However, there are global and encoder-specific block lists that can override certain code points in your allow list. Code points in a block list are always escaped, even if they're included in your allow list.

Global block list

The global block list includes things like private-use characters, control characters, undefined code points, and certain Unicode categories, such as the [Space_Separator category](#), excluding `U+0020 SPACE`. For example, `U+3000 IDEOGRAPHIC SPACE` is escaped even if you specify Unicode range [CJK Symbols and Punctuation \(U+3000-U+303F\)](#) as your allow list.

The global block list is an implementation detail that has changed in every release of .NET. Don't take a dependency on a character being a member of (or not being a member of) the global block list.

Encoder-specific block lists

Examples of encoder-specific blocked code points include `'<'` and `'&'` for the [HTML encoder](#), `'\'` for the [JSON encoder](#), and `'%'` for the [URL encoder](#). For example, the HTML encoder always escapes ampersands (`'&'`), even though the ampersand is in the `BasicLatin` range and all the encoders are initialized with `BasicLatin` by default.

Serialize all characters

To minimize escaping you can use [JavaScriptEncoder.UnsafeRelaxedJsonEscaping](#), as shown in the following example:

C#

```
using System.Text.Encodings.Web;
using System.Text.Json;
using System.Text.Unicode;
```

C#

```
var options3 = new JsonSerializerOptions
{
    Encoder = JavaScriptEncoder.UnsafeRelaxedJsonEscaping,
```

```
        WriteIndented = true
    };
jsonString = JsonSerializer.Serialize(weatherForecast, options3);
```

✖ Caution

Compared to the default encoder, the `UnsafeRelaxedJsonEscaping` encoder is more permissive about allowing characters to pass through unescaped:

- It doesn't escape HTML-sensitive characters such as `<`, `>`, `&`, and `'`.
- It doesn't offer any additional defense-in-depth protections against XSS or information disclosure attacks, such as those which might result from the client and server disagreeing on the `charset`.

Use the unsafe encoder only when it's known that the client will be interpreting the resulting payload as UTF-8 encoded JSON. For example, you can use it if the server is sending the response header `Content-Type: application/json; charset=utf-8`.

Never allow the raw `UnsafeRelaxedJsonEscaping` output to be emitted into an HTML page or a `<script>` element.

See also

- [System.Text.Json overview](#)
- [How to serialize and deserialize JSON](#)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

How to write custom converters for JSON serialization (marshalling) in .NET

Article • 05/23/2023

This article shows how to create custom converters for the JSON serialization classes that are provided in the [System.Text.Json](#) namespace. For an introduction to [System.Text.Json](#), see [How to serialize and deserialize JSON in .NET](#).

A *converter* is a class that converts an object or a value to and from JSON. The [System.Text.Json](#) namespace has built-in converters for most primitive types that map to JavaScript primitives. You can write custom converters:

- To override the default behavior of a built-in converter. For example, you might want `DateTime` values to be represented by mm/dd/yyyy format. By default, ISO 8601-1:2019 is supported, including the RFC 3339 profile. For more information, see [DateTime and DateTimeOffset support in System.Text.Json](#).
- To support a custom value type. For example, a `PhoneNumber` struct.

You can also write custom converters to customize or extend [System.Text.Json](#) with new functionality. The following scenarios are covered later in this article:

- [Deserialize inferred types to object properties](#).
- [Support polymorphic deserialization](#).
- [Support round trip for Stack types](#).
- [Use default system converter](#).

Visual Basic can't be used to write custom converters but can call converters that are implemented in C# libraries. For more information, see [Visual Basic support](#).

Custom converter patterns

There are two patterns for creating a custom converter: the basic pattern and the factory pattern. The factory pattern is for converters that handle type `Enum` or open generics. The basic pattern is for non-generic and closed generic types. For example, converters for the following types require the factory pattern:

- `Dictionary<TKey,TValue>`
- `Enum`
- `List<T>`

Some examples of types that can be handled by the basic pattern include:

- `Dictionary<int, string>`
- `WeekdaysEnum`
- `List<DateTimeOffset>`
- `DateTime`
- `Int32`

The basic pattern creates a class that can handle one type. The factory pattern creates a class that determines, at run time, which specific type is required and dynamically creates the appropriate converter.

Sample basic converter

The following sample is a converter that overrides default serialization for an existing data type. The converter uses mm/dd/yyyy format for `DateTimeOffset` properties.

C#

```
using System.Globalization;
using System.Text.Json;
using System.Text.Json.Serialization;

namespace SystemTextJsonSamples
{
    public class DateTimeOffsetJsonConverter : JsonConverter<DateTimeOffset>
    {
        public override DateTimeOffset Read(
            ref Utf8JsonReader reader,
            Type typeToConvert,
            JsonSerializerOptions options) =>
            DateTimeOffset.ParseExact(reader.GetString()!,
                "MM/dd/yyyy", CultureInfo.InvariantCulture);

        public override void Write(
            Utf8JsonWriter writer,
            DateTimeOffset dateTimeValue,
            JsonSerializerOptions options) =>
            writer.WriteStringValue(dateTimeValue.ToString(
                "MM/dd/yyyy", CultureInfo.InvariantCulture));
    }
}
```

Sample factory pattern converter

The following code shows a custom converter that works with `Dictionary<Enum, TValue>`. The code follows the factory pattern because the first generic type parameter is `Enum`.

and the second is open. The `CanConvert` method returns `true` only for a `Dictionary` with two generic parameters, the first of which is an `Enum` type. The inner converter gets an existing converter to handle whichever type is provided at run time for `TValue`.

C#

```
using System.Reflection;
using System.Text.Json;
using System.Text.Json.Serialization;

namespace SystemTextJsonSamples
{
    public class DictionaryTKeyEnumTValueConverter : JsonConverterFactory
    {
        public override bool CanConvert(Type typeToConvert)
        {
            if (!typeToConvert.IsGenericType)
            {
                return false;
            }

            if (typeToConvert.GetGenericTypeDefinition() !=
typeof(Dictionary<,>))
            {
                return false;
            }

            return typeToConvert.GetGenericArguments()[0].IsEnum;
        }

        public override JsonConverter CreateConverter(
            Type type,
            JsonSerializerOptions options)
        {
            Type keyType = type.GetGenericArguments()[0];
            Type valueType = type.GetGenericArguments()[1];

            JsonConverter converter =
(JsonConverter)Activator.CreateInstance(
            typeof(DictionaryEnumConverterInner<,>).MakeGenericType(
                new Type[] { keyType, valueType }),
            BindingFlags.Instance | BindingFlags.Public,
            binder: null,
            args: new object[] { options },
            culture: null);

            return converter;
        }

        private class DictionaryEnumConverterInner<TKey, TValue> :
            JsonConverter<Dictionary<TKey, TValue>> where TKey : struct,
        Enum
        {

```

```

private readonly JsonConverter< TValue> _valueConverter;
private readonly Type _keyType;
private readonly Type _valueType;

public DictionaryEnumConverterInner(JsonSerializerOptions
options)
{
    // For performance, use the existing converter.
    _valueConverter = (JsonConverter< TValue>)options
        .GetConverter(typeof(TValue));

    // Cache the key and value types.
    _keyType = typeof(TKey);
    _valueType = typeof(TValue);
}

public override Dictionary< TKey, TValue> Read(
    ref Utf8JsonReader reader,
    Type typeToConvert,
    JsonSerializerOptions options)
{
    if (reader.TokenType != JsonTokenType.StartObject)
    {
        throw new JsonException();
    }

    var dictionary = new Dictionary< TKey, TValue>();

    while (reader.Read())
    {
        if (reader.TokenType == JsonTokenType.EndObject)
        {
            return dictionary;
        }

        // Get the key.
        if (reader.TokenType != JsonTokenType.PropertyName)
        {
            throw new JsonException();
        }

        string? propertyName = reader.GetString();

        // For performance, parse with ignoreCase:false first.
        if (!Enum.TryParse(propertyName, ignoreCase: false, out
TKey key) &&
            !Enum.TryParse(propertyName, ignoreCase: true, out
key))
        {
            throw new JsonException(
                $"Unable to convert \'{propertyName}\' to Enum
{(_keyType).ToString()}");
        }

        // Get the value.

```

```
        reader.Read();
        TValue value = _valueConverter.Read(ref reader,
_valueType, options)!;

        // Add to dictionary.
        dictionary.Add(key, value);
    }

    throw new JsonException();
}

public override void Write(
    Utf8JsonWriter writer,
    Dictionary<TKey, TValue> dictionary,
    JsonSerializerOptions options)
{
    writer.WriteStartObject();

    foreach ((TKey key, TValue value) in dictionary)
    {
        var propertyName = key.ToString();
        writer.WritePropertyName

(options.PropertyNamingPolicy?.ConvertName(propertyName) ?? propertyName);

        _valueConverter.Write(writer, value, options);
    }

    writer.WriteEndObject();
}
}
```

Steps to follow the basic pattern

The following steps explain how to create a converter by following the basic pattern:

- Create a class that derives from `JsonConverter<T>` where `T` is the type to be serialized and deserialized.
 - Override the `Read` method to deserialize the incoming JSON and convert it to type `T`. Use the `Utf8JsonReader` that's passed to the method to read the JSON. You don't have to worry about handling partial data, as the serializer passes all the data for the current JSON scope. So it isn't necessary to call `Skip` or `TrySkip` or to validate that `Read` returns `true`.
 - Override the `Write` method to serialize the incoming object of type `T`. Use the `Utf8JsonWriter` that is passed to the method to write the JSON.

- Override the `CanConvert` method only if necessary. The default implementation returns `true` when the type to convert is of type `T`. Therefore, converters that support only type `T` don't need to override this method. For an example of a converter that does need to override this method, see the [polymorphic deserialization](#) section later in this article.

You can refer to the [built-in converters source code](#) as reference implementations for writing custom converters.

Steps to follow the factory pattern

The following steps explain how to create a converter by following the factory pattern:

- Create a class that derives from [JsonConverterFactory](#).
- Override the `CanConvert` method to return `true` when the type to convert is one that the converter can handle. For example, if the converter is for `List<T>`, it might only handle `List<int>`, `List<string>`, and `List<DateTime>`.
- Override the `CreateConverter` method to return an instance of a converter class that will handle the type-to-convert that is provided at run time.
- Create the converter class that the `CreateConverter` method instantiates.

The factory pattern is required for open generics because the code to convert an object to and from a string isn't the same for all types. A converter for an open generic type (`List<T>`, for example) has to create a converter for a closed generic type (`List<DateTime>`, for example) behind the scenes. Code must be written to handle each closed-generic type that the converter can handle.

The `Enum` type is similar to an open generic type: a converter for `Enum` has to create a converter for a specific `Enum` (`WeekdaysEnum`, for example) behind the scenes.

The use of `Utf8JsonReader` in the `Read` method

If your converter is converting a JSON object, the `Utf8JsonReader` will be positioned on the begin object token when the `Read` method begins. You must then read through all the tokens in that object and exit the method with the reader positioned on **the corresponding end object token**. If you read beyond the end of the object, or if you stop before reaching the corresponding end token, you get a `JsonException` exception indicating that:

The converter 'ConverterName' read too much or not enough.

For an example, see the preceding factory pattern sample converter. The `Read` method starts by verifying that the reader is positioned on a start object token. It reads until it finds that it is positioned on the next end object token. It stops on the next end object token because there are no intervening start object tokens that would indicate an object within the object. The same rule about begin token and end token applies if you are converting an array. For an example, see the [Stack<T>](#) sample converter later in this article.

Error handling

The serializer provides special handling for exception types [JsonException](#) and [NotSupportedException](#).

JsonException

If you throw a `JsonException` without a message, the serializer creates a message that includes the path to the part of the JSON that caused the error. For example, the statement `throw new JsonException()` produces an error message like the following example:

Output

```
Unhandled exception. System.Text.Json.JsonException:  
The JSON value could not be converted to System.Object.  
Path: $.Date | LineNumber: 1 | BytePositionInLine: 37.
```

If you do provide a message (for example, `throw new JsonException("Error occurred")`), the serializer still sets the [Path](#), [LineNumber](#), and [BytePositionInLine](#) properties.

NotSupportedException

If you throw a `NotSupportedException`, you always get the path information in the message. If you provide a message, the path information is appended to it. For example, the statement `throw new NotSupportedException("Error occurred.")` produces an error message like the following example:

Output

```
Error occurred. The unsupported member type is located on type  
'System.Collections.Generic.Dictionary`2[Samples.SummaryWords, System.Int32]'
```

```
.
```

```
Path: $.TemperatureRanges | LineNumber: 4 | BytePositionInLine: 24
```

When to throw which exception type

When the JSON payload contains tokens that are not valid for the type being deserialized, throw a `JsonException`.

When you want to disallow certain types, throw a `NotSupportedException`. This exception is what the serializer automatically throws for types that are not supported. For example, `System.Type` is not supported for security reasons, so an attempt to deserialize it results in a `NotSupportedException`.

You can throw other exceptions as needed, but they don't automatically include JSON path information.

Register a custom converter

Register a custom converter to make the `Serialize` and `Deserialize` methods use it. Choose one of the following approaches:

- Add an instance of the converter class to the `JsonSerializerOptions.Converters` collection.
- Apply the `[JsonConverter]` attribute to the properties that require the custom converter.
- Apply the `[JsonConverter]` attribute to a class or a struct that represents a custom value type.

Registration sample - Converters collection

Here's an example that makes the `DateTimeOffsetJsonConverter` the default for properties of type `DateTimeOffset`:

```
C#
```

```
var serializeOptions = new JsonSerializerOptions
{
    WriteIndented = true,
    Converters =
    {
        new DateTimeOffsetJsonConverter()
    }
};
```

```
jsonString = JsonSerializer.Serialize(weatherForecast, serializeOptions);
```

Suppose you serialize an instance of the following type:

C#

```
public class WeatherForecast
{
    public DateTimeOffset Date { get; set; }
    public int TemperatureCelsius { get; set; }
    public string? Summary { get; set; }
}
```

Here's an example of JSON output that shows the custom converter was used:

JSON

```
{
    "Date": "08/01/2019",
    "TemperatureCelsius": 25,
    "Summary": "Hot"
}
```

The following code uses the same approach to deserialize using the custom `DateTimeOffset` converter:

C#

```
var deserializeOptions = new JsonSerializerOptions();
deserializeOptions.Converters.Add(new DateTimeOffsetJsonConverter());
weatherForecast = JsonSerializer.Deserialize<WeatherForecast>(jsonString,
deserializeOptions);
```

Registration sample - [JsonConverter] on a property

The following code selects a custom converter for the `Date` property:

C#

```
public class WeatherForecastWithConverterAttribute
{
    [JsonConverter(typeof(DateTimeOffsetJsonConverter))]
    public DateTimeOffset Date { get; set; }
```

```
    public int TemperatureCelsius { get; set; }
    public string? Summary { get; set; }
}
```

The code to serialize `WeatherForecastWithConverterAttribute` doesn't require the use of `JsonSerializeOptions.Converters`:

```
C#  
  
var serializeOptions = new JsonSerializerOptions
{
    WriteIndented = true
};
jsonString = JsonSerializer.Serialize(weatherForecast, serializeOptions);
```

The code to deserialize also doesn't require the use of `Converters`:

```
C#  
  
weatherForecast =
JsonSerializer.Deserialize<WeatherForecastWithConverterAttribute>
(jsonString);
```

Registration sample - [JsonConverter] on a type

Here's code that creates a struct and applies the `[JsonConverter]` attribute to it:

```
C#  
  
using System.Text.Json.Serialization;  
  
namespace SystemTextJsonSamples
{
    [JsonConverter(typeof(TemperatureConverter))]
    public struct Temperature
    {
        public Temperature(int degrees, bool celsius)
        {
            Degrees = degrees;
            IsCelsius = celsius;
        }

        public int Degrees { get; }
        public bool IsCelsius { get; }
        public bool IsFahrenheit => !IsCelsius;
    }
}
```

```

        public override string ToString() =>
            $"{{Degrees}}{{(IsCelsius ? "C" : "F")}}";

        public static Temperature Parse(string input)
        {
            int degrees = int.Parse(input.Substring(0, input.Length - 1));
            bool celsius = input.Substring(input.Length - 1) == "C";

            return new Temperature(degrees, celsius);
        }
    }
}

```

Here's the custom converter for the preceding struct:

C#

```

using System.Text.Json;
using System.Text.Json.Serialization;

namespace SystemTextJsonSamples
{
    public class TemperatureConverter : JsonConverter<Temperature>
    {
        public override Temperature Read(
            ref Utf8JsonReader reader,
            Type typeToConvert,
            JsonSerializerOptions options) =>
            Temperature.Parse(reader.GetString()!);

        public override void Write(
            Utf8JsonWriter writer,
            Temperature temperature,
            JsonSerializerOptions options) =>
            writer.WriteStringValue(temperature.ToString());
    }
}

```

The `[JsonConverter]` attribute on the struct registers the custom converter as the default for properties of type `Temperature`. The converter is automatically used on the `TemperatureCelsius` property of the following type when you serialize or deserialize it:

C#

```

public class WeatherForecastWithTemperatureStruct
{
    public DateTimeOffset Date { get; set; }
    public Temperature TemperatureCelsius { get; set; }
    public string? Summary { get; set; }
}

```

Converter registration precedence

During serialization or deserialization, a converter is chosen for each JSON element in the following order, listed from highest priority to lowest:

- `[JsonConverter]` applied to a property.
- A converter added to the `Converters` collection.
- `[JsonConverter]` applied to a custom value type or POCO.

If multiple custom converters for a type are registered in the `Converters` collection, the first converter that returns `true` for `CanConvert` is used.

A built-in converter is chosen only if no applicable custom converter is registered.

Converter samples for common scenarios

The following sections provide converter samples that address some common scenarios that built-in functionality doesn't handle.

- [Deserialize inferred types to object properties](#).
- [Support round trip for Stack types](#).
- [Use default system converter](#).

For a sample `DataTable` converter, see [Supported collection types](#).

Deserialize inferred types to object properties

When deserializing to a property of type `object`, a `JsonElement` object is created. The reason is that the deserializer doesn't know what CLR type to create, and it doesn't try to guess. For example, if a JSON property has "true", the deserializer doesn't infer that the value is a `Boolean`, and if an element has "01/01/2019", the deserializer doesn't infer that it's a `DateTime`.

Type inference can be inaccurate. If the deserializer parses a JSON number that has no decimal point as a `long`, that might result in out-of-range issues if the value was originally serialized as a `ulong` or `BigInteger`. Parsing a number that has a decimal point as a `double` might lose precision if the number was originally serialized as a `decimal`.

For scenarios that require type inference, the following code shows a custom converter for `object` properties. The code converts:

- `true` and `false` to `Boolean`

- Numbers without a decimal to `long`
- Numbers with a decimal to `double`
- Dates to `DateTime`
- Strings to `string`
- Everything else to `JsonElement`

C#

```
using System.Text.Json;
using System.Text.Json.Serialization;

namespace CustomConverterInferredTypesToObject
{
    public class ObjectToInferredTypesConverter : JsonConverter<object>
    {
        public override object Read(
            ref Utf8JsonReader reader,
            Type typeToConvert,
            JsonSerializerOptions options) => reader.TokenType switch
        {
            JsonTokenType.True => true,
            JsonTokenType.False => false,
            JsonTokenType.Number when reader.TryGetInt64(out long l) =>
l,
            JsonTokenType.Number => reader.GetDouble(),
            JsonTokenType.String when reader.TryGetDateTime(out DateTime
datetime) => datetime,
            JsonTokenType.String => reader.GetString()!,
            _ => JsonDocument.ParseValue(ref reader).RootElement.Clone()
        };

        public override void Write(
            Utf8JsonWriter writer,
            object objectToWrite,
            JsonSerializerOptions options) =>
            JsonSerializer.Serialize(writer, objectToWrite,
objectToWrite.GetType(), options);
    }

    public class WeatherForecast
    {
        public object? Date { get; set; }
        public object? TemperatureCelsius { get; set; }
        public object? Summary { get; set; }
    }

    public class Program
    {
        public static void Main()
        {
            string jsonString = @"
""Date"": ""2019-08-01T00:00:00-07:00"",
            "Temperature": 65,
            "Summary": "Partly Cloudy"
        }
    }
}
```

```

        "TemperatureCelsius": 25,
        "Summary": "Hot"
    };

    WeatherForecast weatherForecast =
JsonSerializer.Deserialize<WeatherForecast>(jsonString)!;
    Console.WriteLine($"Type of Date property no converter =
{weatherForecast.Date!.GetType()}");

    var options = new JsonSerializerOptions();
    options.WriteIndented = true;
    options.Converters.Add(new ObjectToInferredTypesConverter());
    weatherForecast = JsonSerializer.Deserialize<WeatherForecast>
(jsonString, options)!;
    Console.WriteLine($"Type of Date property with converter =
{weatherForecast.Date!.GetType()}");

    Console.WriteLine(JsonSerializer.Serialize(weatherForecast,
options));
}
}

// Produces output like the following example:
//
//Type of Date property no converter = System.Text.Json.JsonElement
//Type of Date property with converter = System.DateTime
//{
//  "Date": "2019-08-01T00:00:00-07:00",
//  "TemperatureCelsius": 25,
//  "Summary": "Hot"
//}

```

The example shows the converter code and a `WeatherForecast` class with `object` properties. The `Main` method deserializes a JSON string into a `WeatherForecast` instance, first without using the converter, and then using the converter. The console output shows that without the converter, the run-time type for the `Date` property is `JsonElement`; with the converter, the run-time type is `DateTime`.

The [unit tests folder](#) in the `System.Text.Json.Serialization` namespace has more examples of custom converters that handle deserialization to `object` properties.

Support polymorphic deserialization

.NET 7 provides support for both [polymorphic serialization and deserialization](#). However, in previous .NET versions, there was limited polymorphic serialization support and no support for deserialization. If you're using .NET 6 or an earlier version, deserialization requires a custom converter.

Suppose, for example, you have a `Person` abstract base class, with `Employee` and `Customer` derived classes. Polymorphic deserialization means that at design time you can specify `Person` as the deserialization target, and `Customer` and `Employee` objects in the JSON are correctly deserialized at run time. During deserialization, you have to find clues that identify the required type in the JSON. The kinds of clues available vary with each scenario. For example, a discriminator property might be available or you might have to rely on the presence or absence of a particular property. The current release of `System.Text.Json` doesn't provide attributes to specify how to handle polymorphic deserialization scenarios, so custom converters are required.

The following code shows a base class, two derived classes, and a custom converter for them. The converter uses a discriminator property to do polymorphic deserialization. The type discriminator isn't in the class definitions but is created during serialization and is read during deserialization.

ⓘ Important

The example code requires JSON object name/value pairs to stay in order, which is not a standard requirement of JSON.

C#

```
public class Person
{
    public string? Name { get; set; }
}

public class Customer : Person
{
    public decimal CreditLimit { get; set; }
}

public class Employee : Person
{
    public string? OfficeNumber { get; set; }
}
```

C#

```
using System.Text.Json;
using System.Text.Json.Serialization;

namespace SystemTextJsonSamples
{
    public class PersonConverterWithTypeDiscriminator :
        JsonConverter<Person>
```

```
{  
    enum TypeDiscriminator  
    {  
        Customer = 1,  
        Employee = 2  
    }  
  
    public override bool CanConvert(Type typeToConvert) =>  
        typeof(Person).IsAssignableFrom(typeToConvert);  
  
    public override Person Read(  
        ref Utf8JsonReader reader, Type typeToConvert,  
        JsonSerializerOptions options)  
    {  
        if (reader.TokenType != JsonTokenType.StartObject)  
        {  
            throw new JsonException();  
        }  
  
        reader.Read();  
        if (reader.TokenType != JsonTokenType.PropertyName)  
        {  
            throw new JsonException();  
        }  
  
        string? propertyName = reader.GetString();  
        if (propertyName != "TypeDiscriminator")  
        {  
            throw new JsonException();  
        }  
  
        reader.Read();  
        if (reader.TokenType != JsonTokenType.Number)  
        {  
            throw new JsonException();  
        }  
  
        TypeDiscriminator typeDiscriminator =  
(TypeDiscriminator)reader.GetInt32();  
        Person person = typeDiscriminator switch  
        {  
            TypeDiscriminator.Customer => new Customer(),  
            TypeDiscriminator.Employee => new Employee(),  
            _ => throw new JsonException()  
        };  
  
        while (reader.Read())  
        {  
            if (reader.TokenType == JsonTokenType.EndObject)  
            {  
                return person;  
            }  
  
            if (reader.TokenType == JsonTokenType.PropertyName)  
            {  
                throw new JsonException();  
            }  
        }  
    }  
}
```

```

        propertyName = reader.GetString();
        reader.Read();
        switch (propertyName)
        {
            case "CreditLimit":
                decimal creditLimit = reader.GetDecimal();
                ((Customer)person).CreditLimit = creditLimit;
                break;
            case "OfficeNumber":
                string? officeNumber = reader.GetString();
                ((Employee)person).OfficeNumber = officeNumber;
                break;
            case "Name":
                string? name = reader.GetString();
                person.Name = name;
                break;
        }
    }

    throw new JsonException();
}

public override void Write(
    Utf8JsonWriter writer, Person person, JsonSerializerOptions
options)
{
    writer.WriteStartObject();

    if (person is Customer customer)
    {
        writer.WriteNumber("TypeDiscriminator",
(int)TypeDiscriminator.Customer);
        writer.WriteNumber("CreditLimit", customer.CreditLimit);
    }
    else if (person is Employee employee)
    {
        writer.WriteNumber("TypeDiscriminator",
(int)TypeDiscriminator.Employee);
        writer.WriteString("OfficeNumber", employee.OfficeNumber);
    }

    writer.WriteString("Name", person.Name);

    writer.WriteEndObject();
}
}
}

```

The following code registers the converter:

C#

```
var serializeOptions = new JsonSerializerOptions();
serializeOptions.Converters.Add(new PersonConverterWithTypeDiscriminator());
```

The converter can deserialize JSON that was created by using the same converter to serialize, for example:

JSON

```
[  
  {  
    "TypeDiscriminator": 1,  
    "CreditLimit": 10000,  
    "Name": "John"  
  },  
  {  
    "TypeDiscriminator": 2,  
    "OfficeNumber": "555-1234",  
    "Name": "Nancy"  
  }  
]
```

The converter code in the preceding example reads and writes each property manually. An alternative is to call `Deserialize` or `Serialize` to do some of the work. For an example, see [this StackOverflow post](#).

An alternative way to do polymorphic deserialization

You can call `Deserialize` in the `Read` method:

- Make a clone of the `Utf8JsonReader` instance. Since `Utf8JsonReader` is a struct, this just requires an assignment statement.
- Use the clone to read through the discriminator tokens.
- Call `Deserialize` using the original `Reader` instance once you know the type you need. You can call `Deserialize` because the original `Reader` instance is still positioned to read the begin object token.

A disadvantage of this method is you can't pass in the original options instance that registers the converter to `Deserialize`. Doing so would cause a stack overflow, as explained in [Required properties](#). The following example shows a `Read` method that uses this alternative:

C#

```

public override Person Read(
    ref Utf8JsonReader reader, Type typeToConvert, JsonSerializerOptions
options)
{
    Utf8JsonReader readerClone = reader;

    if (readerClone.TokenType != JsonTokenType.StartObject)
    {
        throw new JsonException();
    }

    readerClone.Read();
    if (readerClone.TokenType != JsonTokenType.PropertyName)
    {
        throw new JsonException();
    }

    string? propertyName = readerClone.GetString();
    if (propertyName != "TypeDiscriminator")
    {
        throw new JsonException();
    }

    readerClone.Read();
    if (readerClone.TokenType != JsonTokenType.Number)
    {
        throw new JsonException();
    }

    TypeDiscriminator typeDiscriminator =
(TypeDiscriminator)readerClone.GetInt32();
    Person person = typeDiscriminator switch
    {
        TypeDiscriminator.Customer => JsonSerializer.Deserialize<Customer>
(ref reader)!, 
        TypeDiscriminator.Employee => JsonSerializer.Deserialize<Employee>
(ref reader)!, 
        _ => throw new JsonException()
    };
    return person;
}

```

Support round trip for `Stack` types

If you deserialize a JSON string into a `Stack` object and then serialize that object, the contents of the stack are in reverse order. This behavior applies to the following types and interfaces, and user-defined types that derive from them:

- `Stack`
- `Stack<T>`

- [ConcurrentStack<T>](#)
- [ImmutableStack<T>](#)
- [IImmutableStack<T>](#)

To support serialization and deserialization that retains the original order in the stack, a custom converter is required.

The following code shows a custom converter that enables round-tripping to and from `Stack<T>` objects:

C#

```
using System.Diagnostics;
using System.Reflection;
using System.Text.Json;
using System.Text.Json.Serialization;

namespace SystemTextJsonSamples
{
    public class JsonConverterFactoryForStackOfT : JsonConverterFactory
    {
        public override bool CanConvert(Type typeToConvert)
            => typeToConvert.IsGenericType
            && typeToConvert.GetGenericTypeDefinition() == typeof(Stack<>);

        public override JsonConverter CreateConverter(
            Type typeToConvert, JsonSerializerOptions options)
        {
            Debug.Assert(typeToConvert.IsGenericType &&
                typeToConvert.GetGenericTypeDefinition() ==
typeof(Stack<>));

            Type elementType = typeToConvert.GetGenericArguments()[0];

            JsonConverter converter =
(JsonConverter)Activator.CreateInstance(
    typeof(JsonConverterForStackOfT<>)
        .MakeGenericType(new Type[] { elementType }),
    BindingFlags.Instance | BindingFlags.Public,
    binder: null,
    args: null,
    culture: null)!;

            return converter;
        }
    }

    public class JsonConverterForStackOfT<T> : JsonConverter<Stack<T>>
    {
        public override Stack<T> Read(
            ref Utf8JsonReader reader, Type typeToConvert,
            JsonSerializerOptions options)
```

```

    {
        if (reader.TokenType != JsonTokenType.StartArray)
        {
            throw new JsonException();
        }
        reader.Read();

        var elements = new Stack<T>();

        while (reader.TokenType != JsonTokenType.EndArray)
        {
            elements.Push(JsonSerializer.Deserialize<T>(ref reader,
options)!);

            reader.Read();
        }

        return elements;
    }

    public override void Write(
        Utf8JsonWriter writer, Stack<T> value, JsonSerializerOptions
options)
{
    writer.WriteStartArray();

    var reversed = new Stack<T>(value);

    foreach (T item in reversed)
    {
        JsonSerializer.Serialize(writer, item, options);
    }

    writer.WriteEndArray();
}
}
}

```

The following code registers the converter:

```

C#

var options = new JsonSerializerOptions
{
    Converters = { new JsonConverterFactoryForStackOfT() },
};

```

Use default system converter

In some scenarios, you might want to use the default system converter in a custom converter. To do that, get the system converter from the `JsonSerializerOptions.Default` property, as shown in the following example:

C#

```
public class MyCustomConverter : JsonConverter<int>
{
    private readonly static JsonConverter<int> s_defaultConverter =
        (JsonConverter<int>)JsonSerializerOptions.Default.GetConverter(typeof(int));

    // Custom serialization logic
    public override void Write(
        Utf8JsonWriter writer, int value, JsonSerializerOptions options)
    {
        writer.WriteStringValue(value.ToString());
    }

    // Fall back to default deserialization logic
    public override int Read(
        ref Utf8JsonReader reader, Type typeToConvert, JsonSerializerOptions options)
    {
        return s_defaultConverter.Read(ref reader, typeToConvert, options);
    }
}
```

Handle null values

By default, the serializer handles null values as follows:

- For reference types and `Nullable<T>` types:
 - It does not pass `null` to custom converters on serialization.
 - It does not pass `JsonTokenType.Null` to custom converters on deserialization.
 - It returns a `null` instance on deserialization.
 - It writes `null` directly with the writer on serialization.
- For non-nullable value types:
 - It passes `JsonTokenType.Null` to custom converters on deserialization. (If no custom converter is available, a `JsonException` exception is thrown by the internal converter for the type.)

This null-handling behavior is primarily to optimize performance by skipping an extra call to the converter. In addition, it avoids forcing converters for nullable types to check for `null` at the start of every `Read` and `Write` method override.

To enable a custom converter to handle `null` for a reference or value type, override `JsonConverter<T>.HandleNull` to return `true`, as shown in the following example:

C#

```
using System.Text.Json;
using System.Text.Json.Serialization;

namespace CustomConverterHandleNull
{
    public class Point
    {
        public int X { get; set; }
        public int Y { get; set; }

        [JsonConverter(typeof(DescriptionConverter))]
        public string? Description { get; set; }
    }

    public class DescriptionConverter : JsonConverter<string>
    {
        public override bool HandleNull => true;

        public override string Read(
            ref Utf8JsonReader reader,
            Type typeToConvert,
            JsonSerializerOptions options) =>
            reader.GetString() ?? "No description provided.";

        public override void Write(
            Utf8JsonWriter writer,
            string value,
            JsonSerializerOptions options) =>
            writer.WriteStringValue(value);
    }
}

public class Program
{
    public static void Main()
    {
        string json = @"""x"":1,""y"":2,""Description"":null};

        Point point = JsonSerializer.Deserialize<Point>(json)!;
        Console.WriteLine($"Description: {point.Description}");
    }
}

// Produces output like the following example:
// 
//Description: No description provided.
```

Preserve references

By default, reference data is only cached for each call to `Serialize` or `Deserialize`. To persist references from one `Serialize/Deserialize` call to another one, root the `ReferenceResolver` instance in the call site of `Serialize/Deserialize`. The following code shows an example for this scenario:

- You write a custom converter for the `Company` type.
- You don't want to manually serialize the `Supervisor` property, which is an `Employee`. You want to delegate that to the serializer and you also want to preserve the references that you have already saved.

Here are the `Employee` and `Company` classes:

```
C#  
  
public class Employee  
{  
    public string? Name { get; set; }  
    public Employee? Manager { get; set; }  
    public List<Employee>? DirectReports { get; set; }  
    public Company? Company { get; set; }  
}  
  
public class Company  
{  
    public string? Name { get; set; }  
    public Employee? Supervisor { get; set; }  
}
```

The converter looks like this:

```
C#  
  
class CompanyConverter : JsonConverter<Company>  
{  
    public override Company Read(ref Utf8JsonReader reader, Type  
typeToConvert, JsonSerializerOptions options)  
    {  
        throw new NotImplementedException();  
    }  
  
    public override void Write(Utf8JsonWriter writer, Company value,  
JsonSerializerOptions options)  
    {  
        writer.WriteStartObject();  
  
        writer.WriteString("Name", value.Name);  
    }  
}
```

```

        writer.WritePropertyName("Supervisor");
        JsonSerializer.Serialize(writer, value.Supervisor, options);

        writer.WriteEndObject();
    }
}

```

A class that derives from [ReferenceResolver](#) stores the references in a dictionary:

C#

```

class MyReferenceResolver : ReferenceResolver
{
    private uint _referenceCount;
    private readonly Dictionary<string, object> _referenceIdToObjectMap =
    new ();
    private readonly Dictionary<object, string> _objectToReferenceIdMap =
    new (ReferenceEqualityComparer.Instance);

    public override void AddReference(string referenceId, object value)
    {
        if (!_referenceIdToObjectMap.TryAdd(referenceId, value))
        {
            throw new JsonException();
        }
    }

    public override string GetReference(object value, out bool
alreadyExists)
    {
        if (_objectToReferenceIdMap.TryGetValue(value, out string?
referenceId))
        {
            alreadyExists = true;
        }
        else
        {
            _referenceCount++;
            referenceId = _referenceCount.ToString();
            _objectToReferenceIdMap.Add(value, referenceId);
            alreadyExists = false;
        }
    }

    return referenceId;
}

public override object ResolveReference(string referenceId)
{
    if (!_referenceIdToObjectMap.TryGetValue(referenceId, out object?
value))
    {
        throw new JsonException();
    }
}

```

```
        }

        return value;
    }
}
```

A class that derives from [ReferenceHandler](#) holds an instance of [MyReferenceResolver](#) and creates a new instance only when needed (in a method named `Reset` in this example):

C#

```
class MyReferenceHandler : ReferenceHandler
{
    public MyReferenceHandler() => Reset();

    private ReferenceResolver? _rootedResolver;
    public override ReferenceResolver CreateResolver() => _rootedResolver!;
    public void Reset() => _rootedResolver = new MyReferenceResolver();

}
```

When the sample code calls the serializer, it uses a [JsonSerializerOptions](#) instance in which the [ReferenceHandler](#) property is set to an instance of [MyReferenceHandler](#). When you follow this pattern, be sure to reset the [ReferenceResolver](#) dictionary when you're finished serializing, to keep it from growing forever.

C#

```
var options = new JsonSerializerOptions();

options.Converters.Add(new CompanyConverter());
var myReferenceHandler = new MyReferenceHandler();
options.ReferenceHandler = myReferenceHandler;
options.DefaultIgnoreCondition = JsonIgnoreCondition.WhenWritingNull;
options.WriteIndented = true;

string str = JsonSerializer.Serialize(tyler, options);

// Reset after serializing to avoid out of bounds memory growth in the
// resolver.
myReferenceHandler.Reset();
```

The preceding example only does serialization, but a similar approach can be adopted for deserialization.

Other custom converter samples

The [Migrate from Newtonsoft.Json to System.Text.Json](#) article contains additional samples of custom converters.

The [unit tests folder ↗](#) in the `System.Text.Json.Serialization` source code includes other custom converter samples, such as:

- [Int32 converter that converts null to 0 on deserialize ↗](#)
- [Int32 converter that allows both string and number values on deserialize ↗](#)
- [Enum converter ↗](#)
- [List<T> converter that accepts external data ↗](#)
- [Long\[\] converter that works with a comma-delimited list of numbers ↗](#)

If you need to make a converter that modifies the behavior of an existing built-in converter, you can get [the source code of the existing converter ↗](#) to serve as a starting point for customization.

Additional resources

- [Source code for built-in converters ↗](#)
- [System.Text.Json overview](#)
- [How to serialize and deserialize JSON](#)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



.NET feedback

.NET is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

Customize a JSON contract

Article • 06/16/2023

The [System.Text.Json](#) library constructs a JSON *contract* for each .NET type, which defines how the type should be serialized and deserialized. The contract is derived from the type's shape, which includes characteristics such as its properties and fields and whether it implements the [IEnumerable](#) or [IDictionary](#) interface. Types are mapped to contracts either at run time using reflection or at compile time using the source generator.

Starting in .NET 7, you can customize these JSON contracts to provide more control over how types are converted into JSON and vice versa. The following list shows just some examples of the types of customizations you can make to serialization and deserialization:

- Serialize private fields and properties.
- Support multiple names for a single property (for example, if a previous library version used a different name).
- Ignore properties with a specific name, type, or value.
- Distinguish between explicit `null` values and the lack of a value in the JSON payload.
- Support [System.Runtime.Serialization](#) attributes, such as [DataContractAttribute](#). For more information, see [System.Runtime.Serialization attributes](#).
- Throw an exception if the JSON includes a property that's not part of the target type. For more information, see [Handle missing members](#).

How to opt in

There are two ways to plug into customization. Both involve obtaining a resolver, whose job is to provide a [JsonTypeInfo](#) instance for each type that needs to be serialized.

- By calling the [DefaultJsonTypeInfoResolver\(\)](#) constructor to obtain the [JsonSerializerOptions.TypeInfoResolver](#) and adding your [custom actions](#) to its [Modifiers](#) property.

For example:

```
C#  
  
JsonSerializerOptions options = new()  
{  
    TypeInfoResolver = new DefaultJsonTypeInfoResolver
```

```

    {
        Modifiers =
        {
            MyCustomModifier1,
            MyCustomModifier2
        }
    };
}

```

If you add multiple modifiers, they'll be called sequentially.

- By writing a custom resolver that implements [IJsonTypeInfoResolver](#).
 - If a type isn't handled, [IJsonTypeInfoResolver.GetTypeInfo](#) should return `null` for that type.
 - You can also combine your custom resolver with others, for example, the default resolver. The resolvers will be queried in order until a non-null [JsonTypeInfo](#) value is returned for the type.

Configurable aspects

The [JsonTypeInfo.Kind](#) property indicates how the converter serializes a given type—for example, as an object or as an array, and whether its properties are serialized. You can query this property to determine which aspects of a type's JSON contract you can configure. There are four different kinds:

| <code>JsonTypeInfo.Kind</code> | Description |
|---|---|
| JsonTypeInfoKind.Object | The converter will serialize the type into a JSON object and uses its properties. This kind is used for most class and struct types and allows for the most flexibility. |
| JsonTypeInfoKind.Enumerable | The converter will serialize the type into a JSON array. This kind is used for types like <code>List<T></code> and array. |
| JsonTypeInfoKind.Dictionary | The converter will serialize the type into a JSON object. This kind is used for types like <code>Dictionary<K, V></code> . |
| JsonTypeInfoKind.None | The converter doesn't specify how it will serialize the type or what <code>JsonTypeInfo</code> properties it will use. This kind is used for types like System.Object , <code>int</code> , and <code>string</code> , and for all types that use a custom converter. |

Modifiers

A modifier is an `Action<JsonTypeInfo>` or a method with a `JsonTypeInfo` parameter that gets the current state of the contract as an argument and makes modifications to the contract. For example, you could iterate through the prepopulated properties on the specified `JsonTypeInfo` to find the one you're interested in and then modify its `JsonPropertyInfo.Get` property (for serialization) or `JsonPropertyInfo.Set` property (for deserialization). Or, you can construct a new property using `JsonTypeInfo.CreateJsonPropertyInfo(Type, String)` and add it to the `JsonTypeInfo.Properties` collection.

The following table shows the modifications you can make and how to achieve them.

| Modification | Applicable <code>JsonTypeInfo.Kind</code> | How to achieve it | Example |
|---|--|---|--|
| Customize a property's value | <code>JsonTypeInfoKind.Object</code> | Modify the <code>JsonPropertyInfo.Get</code> delegate (for serialization) or <code>JsonPropertyInfo.Set</code> delegate (for deserialization) for the property. | Increment a property's value |
| Add or remove properties | <code>JsonTypeInfoKind.Object</code> | Add or remove items from the <code>JsonTypeInfo.Properties</code> list. | Serialize private fields |
| Conditionally serialize a property | <code>JsonTypeInfoKind.Object</code> | Modify the <code>JsonPropertyInfo.ShouldSerialize</code> predicate for the property. | Ignore properties with a specific type |
| Customize number handling for a specific type | <code>JsonTypeInfoKind.None</code> | Modify the <code>JsonTypeInfo.NumberHandling</code> value for the type. | Allow int values to be strings |

Example: Increment a property's value

Consider the following example where the modifier increments the value of a certain property on deserialization by modifying its `JsonPropertyInfo.Set` delegate. Besides defining the modifier, the example also introduces a new attribute that it uses to locate the property whose value should be incremented. This is an example of *customizing a property*.

C#

```
using System.Text.Json;
using System.Text.Json.Serialization.Metadata;
```

```

namespace Serialization
{
    // Custom attribute to annotate the property
    // we want to be incremented.
    [AttributeUsage(AttributeTargets.Property)]
    class SerializationCountAttribute : Attribute
    {
    }

    // Example type to serialize and deserialize.
    class Product
    {
        public string Name { get; set; } = "";
        [SerializationCount]
        public int RoundTrips { get; set; }
    }

    public class SerializationCountExample
    {
        // Custom modifier that increments the value
        // of a specific property on deserialization.
        static void IncrementCounterModifier(JsonTypeInfo typeInfo)
        {
            foreach (JsonPropertyInfo propertyInfo in typeInfo.Properties)
            {
                if (propertyInfo.PropertyType != typeof(int))
                    continue;

                object[] serializationCountAttributes =
                    propertyInfo.AttributeProvider?.GetCustomAttributes(typeof(SerializationCountAttribute), true) ?? Array.Empty<object>();
                SerializationCountAttribute? attribute =
                    serializationCountAttributes.Length == 1 ?
                    (SerializationCountAttribute)serializationCountAttributes[0] : null;

                if (attribute != null)
                {
                    Action<object, object?>? setProperty = propertyInfo.Set;
                    if (setProperty is not null)
                    {
                        propertyInfo.Set = (obj, value) =>
                        {
                            if (value != null)
                            {
                                // Increment the value by 1.
                                value = (int)value + 1;
                            }
                            setProperty (obj, value);
                        };
                    }
                }
            }
        }
    }
}

```

```

public static void RunIt()
{
    var product = new Product
    {
        Name = "Aquafresh"
    };

    JsonSerializerOptions options = new()
    {
        TypeInfoResolver = new DefaultJsonTypeInfoResolver
        {
            Modifiers = { IncrementCounterModifier }
        }
    };

    // First serialization and deserialization.
    string serialized = JsonSerializer.Serialize(product, options);
    Console.WriteLine(serialized);
    // {"Name":"Aquafresh","RoundTrips":0}

    Product deserialized = JsonSerializer.Deserialize<Product>
(serialized, options)!;
    Console.WriteLine($"{deserialized.RoundTrips}");
    // 1

    // Second serialization and deserialization.
    serialized = JsonSerializer.Serialize(deserialized, options);
    Console.WriteLine(serialized);
    // {"Name":"Aquafresh","RoundTrips":1}

    deserialized = JsonSerializer.Deserialize<Product>(serialized,
options)!;
    Console.WriteLine($"{deserialized.RoundTrips}");
    // 2
}
}

```

Notice in the output that the value of `RoundTrips` is incremented each time the `Product` instance is deserialized.

Example: Serialize private fields

By default, `System.Text.Json` ignores private fields and properties. This example adds a new class-wide attribute, `JsonIncludePrivateFieldsAttribute`, to change that default. If the modifier finds the attribute on a type, it adds all the private fields on the type as new properties to `JsonTypeInfo`.

```
using System.Reflection;
using System.Text.Json;
using System.Text.Json.Serialization;
using System.Text.Json.Serialization.Metadata;

namespace Serialization
{
    [AttributeUsage(AttributeTargets.Class | AttributeTargets.Struct)]
    public class JsonIncludePrivateFieldsAttribute : Attribute { }

    [JsonIncludePrivateFields]
    public class Human
    {
        private string _name;
        private int _age;

        public Human()
        {
            // This constructor should be used only by deserializers.
            _name = null!;
            _age = 0;
        }

        public static Human Create(string name, int age)
        {
            Human h = new()
            {
                _name = name,
                _age = age
            };

            return h;
        }

        [JsonIgnore]
        public string Name
        {
            get => _name;
            set => throw new NotSupportedException();
        }

        [JsonIgnore]
        public int Age
        {
            get => _age;
            set => throw new NotSupportedException();
        }
    }

    public class PrivateFieldsExample
    {
        static void AddPrivateFieldsModifier(JsonTypeInfo jsonTypeInfo)
        {
            if (jsonTypeInfo.Kind != JsonTypeInfoKind.Object)
```

```

        return;

        if
(!jsonTypeInfo.Type.isDefined(typeof(JsonIncludePrivateFieldsAttribute),
inherit: false))
        return;

        foreach (FieldInfo field in
jsonTypeInfo.Type.GetFields(BindingFlags.Instance | BindingFlags.NonPublic))
{
    JsonPropertyInfo json PropertyInfo =
jsonTypeInfo.CreateJsonPropertyInfo(field.FieldType, field.Name);
    json PropertyInfo.Get = field.GetValue;
    json PropertyInfo.Set = field.SetValue;

    jsonTypeInfo.Properties.Add(json PropertyInfo);
}
}

public static void RunIt()
{
    var options = new JsonSerializerOptions
{
    TypeInfoResolver = new DefaultJsonTypeInfoResolver
    {
        Modifiers = { AddPrivateFieldsModifier }
    }
};

var human = Human.Create("Julius", 37);
string json = JsonSerializer.Serialize(human, options);
Console.WriteLine(json);
// {"_name":"Julius","_age":37}

Human deserializedHuman = JsonSerializer.Deserialize<Human>
(json, options)!;
Console.WriteLine($"[Name={deserializedHuman.Name}; Age=
{deserializedHuman.Age}]");
// [Name=Julius; Age=37]
}
}
}

```

Tip

If your private field names start with underscores, consider removing the underscores from the names when you add the fields as new JSON properties.

Example: Ignore properties with a specific type

Perhaps your model has properties with specific names or types that you don't want to expose to users. For example, you might have a property that stores credentials or some information that's useless to have in the payload.

The following example shows how to filter out properties with a specific type, `SecretHolder`. It does this by using an `IList<T>` extension method to remove any properties that have the specified type from the `JsonTypeInfo.Properties` list. The filtered properties completely disappear from the contract, which means `System.Text.Json` doesn't look at them either during serialization or deserialization.

C#

```
using System.Text.Json;
using System.Text.Json.Serialization.Metadata;

namespace Serialization
{
    class ExampleClass
    {
        public string Name { get; set; } = "";
        public SecretHolder? Secret { get; set; }
    }

    class SecretHolder
    {
        public string Value { get; set; } = "";
    }

    class IgnorePropertiesWithType
    {
        private readonly Type[] _ignoredTypes;

        public IgnorePropertiesWithType(params Type[] ignoredTypes)
            => _ignoredTypes = ignoredTypes;

        public void ModifyTypeInfo(JsonTypeInfo ti)
        {
            if (ti.Kind != JsonTypeInfoKind.Object)
                return;

            ti.Properties.RemoveAll(prop =>
                _ignoredTypes.Contains(prop.PropertyType));
        }
    }

    public class IgnoreTypeExample
    {
        public static void RunIt()
        {
            var modifier = new
                IgnorePropertiesWithType(typeof(SecretHolder));
        }
    }
}
```

```

JsonSerializerOptions options = new()
{
    TypeInfoResolver = new DefaultJsonTypeInfoResolver
    {
        Modifiers = { modifier.ModifyTypeInfo }
    }
};

ExampleClass obj = new()
{
    Name = "Password",
    Secret = new SecretHolder { Value = "MySecret" }
};

string output = JsonSerializer.Serialize(obj, options);
Console.WriteLine(output);
// {"Name": "Password"}
}

public static class ListHelpers
{
    // IList<T> implementation of List<T>.RemoveAll method.
    public static void RemoveAll<T>(this IList<T> list, Predicate<T>
predicate)
    {
        for (int i = 0; i < list.Count; i++)
        {
            if (predicate(list[i]))
            {
                list.RemoveAt(i--);
            }
        }
    }
}
}

```

Example: Allow int values to be strings

Perhaps your input JSON can contain quotes around one of the numeric types but not on others. If you had control over the class, you could place [JsonNumberHandlingAttribute](#) on the type to fix this, but you don't. Before .NET 7, you'd need to write a [custom converter](#) to fix this behavior, which requires writing a fair bit of code. Using contract customization, you can customize the number handling behavior for any type.

The following example changes the behavior for all `int` values. The example can be easily adjusted to apply to any type or for a specific property of any type.

C#

```
using System.Text.Json;
using System.Text.Json.Serialization;
using System.Text.Json.Serialization.Metadata;

namespace Serialization
{
    public class Point
    {
        public int X { get; set; }
        public int Y { get; set; }
    }

    public class AllowIntsAsStringsExample
    {
        static void SetNumberHandlingModifier(JsonTypeInfo jsonTypeInfo)
        {
            if (jsonTypeInfo.Type == typeof(int))
            {
                jsonTypeInfo.NumberHandling =
JsonNumberHandling.AllowReadingFromString;
            }
        }

        public static void RunIt()
        {
            JsonSerializerOptions options = new()
            {
                TypeInfoResolver = new DefaultJsonTypeInfoResolver
                {
                    Modifiers = { SetNumberHandlingModifier }
                }
            };

            // Triple-quote syntax is a C# 11 feature.
            Point point = JsonSerializer.Deserialize<Point>("""
{ "X": "12", "Y": "3" }""", options)!;
            Console.WriteLine($"({point.X},{point.Y})");
            // (12,3)
        }
    }
}
```

Without the modifier to allow reading `int` values from a string, the program would have ended with an exception:

Unhandled exception. System.Text.Json.JsonException: The JSON value could not be converted to System.Int32. Path: \$.X | LineNumber: 0 | BytePositionInLine: 9.

Other ways to customize serialization

Besides customizing a contract, there are other ways to influence serialization and deserialization behavior, including the following:

- By using attributes derived from [JsonAttribute](#), for example, [JsonIgnoreAttribute](#) and [JsonPropertyOrderAttribute](#).
- By modifying [JsonSerializerOptions](#), for example, to set a naming policy or serialize enumeration values as strings instead of numbers.
- By writing a custom converter that does the actual work of writing the JSON and, during deserialization, constructing an object.

Contract customization is an improvement over these pre-existing customizations because you might not have access to the type to add attributes, and writing a custom converter is complex and hurts performance.

See also

- [JSON contract customization \(blog post\)](#) ↗
- [What's new in System.Text.Json in .NET 7 \(blog post\)](#) ↗

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

XML and SOAP serialization

Article • 04/05/2023

XML serialization converts (serializes) the public fields and properties of an object, and the parameters and return values of methods, into an XML stream that conforms to a specific XML Schema definition language (XSD) document. XML serialization results in strongly typed classes with public properties and fields that are converted to a serial format (in this case, XML) for storage or transport.

Because XML is an open standard, the XML stream can be processed by any application, as needed, regardless of platform. For example, XML Web services created using ASP.NET use the [XmlSerializer](#) class to create XML streams that pass data between XML Web service applications throughout the Internet or on intranets. Conversely, deserialization takes such an XML stream and reconstructs the object.

XML serialization can also be used to serialize objects into XML streams that conform to the SOAP specification. SOAP is a protocol based on XML, designed specifically to transport procedure calls using XML.

To serialize or deserialize objects, use the [XmlSerializer](#) class. To create the classes to be serialized, use the XML Schema Definition tool.

See also

- [Binary Serialization](#)
- [XML Web Services created using ASP.NET and XML Web Service clients](#)



Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



.NET feedback

The .NET documentation is open source. Provide feedback here.



[Open a documentation issue](#)



[Provide product feedback](#)

XML serialization

Article • 04/05/2023

Serialization is the process of converting an object into a form that can be readily transported. For example, you can serialize an object and transport it over the Internet using HTTP between a client and a server. On the other end, deserialization reconstructs the object from the stream.

XML serialization serializes only the public fields and property values of an object into an XML stream. XML serialization does not include type information. For example, if you have a **Book** object that exists in the **Library** namespace, there is no guarantee that it is deserialized into an object of the same type.

ⓘ Note

XML serialization does not convert methods, indexers, private fields, or read-only properties (except read-only collections). To serialize all an object's fields and properties, both public and private, use the **DataContractSerializer** instead of XML serialization.

The central class in XML serialization is the **XmlSerializer** class, and the most important methods in this class are the **Serialize** and **Deserialize** methods. The **XmlSerializer** creates C# files and compiles them into .dll files to perform this serialization. The [XML Serializer Generator Tool \(Sgen.exe\)](#) is designed to generate these serialization assemblies in advance to be deployed with your application and improve startup performance. The XML stream generated by the **XmlSerializer** is compliant with the World Wide Web Consortium (W3C) [XML Schema definition language \(XSD\) 1.0 recommendation](#). Furthermore, the data types generated are compliant with the document titled "XML Schema Part 2: Datatypes."

The data in your objects is described using programming language constructs like classes, fields, properties, primitive types, arrays, and even embedded XML in the form of **XmlElement** or **XmlAttribute** objects. You have the option of creating your own classes, annotated with attributes, or using the XML Schema Definition tool to generate the classes based on an existing XML Schema.

If you have an XML Schema, you can run the XML Schema Definition tool to produce a set of classes that are strongly typed to the schema and annotated with attributes. When an instance of such a class is serialized, the generated XML adheres to the XML Schema. Provided with such a class, you can program against an easily manipulated

object model while being assured that the generated XML conforms to the XML schema. This is an alternative to using other classes in .NET, such as the **XmlReader** and **XmlWriter** classes, to parse and write an XML stream. For more information, see [XML Documents and Data](#). These classes allow you to parse any XML stream. In contrast, use the **XmlSerializer** when the XML stream is expected to conform to a known XML Schema.

Attributes control the XML stream generated by the **XmlSerializer** class, allowing you to set the XML namespace, element name, attribute name, and so on, of the XML stream. For more information about these attributes and how they control XML serialization, see [Controlling XML Serialization Using Attributes](#). For a table of those attributes that are used to control the generated XML, see [Attributes That Control XML Serialization](#).

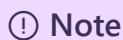
The **XmlSerializer** class can further serialize an object and generate an encoded SOAP XML stream. The generated XML adheres to section 5 of the World Wide Web Consortium document titled "Simple Object Access Protocol (SOAP) 1.1." For more information about this process, see [How to: Serialize an Object as a SOAP-Encoded XML Stream](#). For a table of the attributes that control the generated XML, see [Attributes That Control Encoded SOAP Serialization](#).

The **XmlSerializer** class generates the SOAP messages created by, and passed to, XML Web services. To control the SOAP messages, you can apply attributes to the classes, return values, parameters, and fields found in an XML Web service file (.asmx). You can use both the attributes listed in "Attributes That Control XML Serialization" and "Attributes That Control Encoded SOAP Serialization" because an XML Web service can use either the literal or encoded SOAP style. For more information about using attributes to control the XML generated by an XML Web service, see [XML Serialization with XML Web Services](#). For more information about SOAP and XML Web services, see [Customizing SOAP Message Formatting](#).

Security Considerations for **XmlSerializer** Applications

When creating an application that uses the **XmlSerializer**, be aware of the following items and their implications:

- The **XmlSerializer** creates C# (.cs) files and compiles them into .dll files in the directory named by the TEMP environment variable; serialization occurs with those DLLs.



Note

These serialization assemblies can be generated in advance and signed by using the SGGen.exe tool. This does not work on a server of Web services. In other words, it is only for client use and for manual serialization.

The code and the DLLs are vulnerable to a malicious process at the time of creation and compilation. It might be possible for two or more users to share the TEMP directory. Sharing a TEMP directory is dangerous if the two accounts have different security privileges and the higher-privilege account runs an application using the **XmlSerializer**. In this case, one user can breach the computer's security by replacing either the .cs or .dll file that is compiled. To eliminate this concern, always be sure that each account on the computer has its own profile. By default, the TEMP environment variable points to a different directory for each account.

- If a malicious user sends a continuous stream of XML data to a Web server (a denial of service attack), then the **XmlSerializer** continues to process the data until the computer runs low on resources.

This kind of attack is eliminated if you are using a computer running Internet Information Services (IIS), and your application is running within IIS. IIS features a gate that does not process streams longer than a set amount (the default is 4 KB). If you create an application that does not use IIS and deserializes with the **XmlSerializer**, you should implement a similar gate that prevents a denial of service attack.

- The **XmlSerializer** serializes data and runs any code using any type given to it.

There are two ways in which a malicious object presents a threat. It could run malicious code or it could inject malicious code into the C# file created by the **XmlSerializer**. In the second case, there is a theoretical possibility that a malicious object may somehow inject code into the C# file created by the **XmlSerializer**. Although this issue has been examined thoroughly, and such an attack is considered unlikely, you should take the precaution of never serializing data with an unknown and untrusted type.

- Serialized sensitive data might be vulnerable.

After the **XmlSerializer** has serialized data, it can be stored as an XML file or other data store. If your data store is available to other processes, or is visible on an intranet or the Internet, the data can be stolen and used maliciously. For example, if you create an application that serializes orders that include credit card numbers, the data is highly sensitive. To help prevent this, always protect the store for your data and take steps to keep it private.

Serialization of a Simple Class

The following code example shows a basic class with a public field.

C#

```
public class OrderForm
{
    public DateTime OrderDate;
}
```

When an instance of this class is serialized, it might resemble the following.

XML

```
<OrderForm>
    <OrderDate>12/12/01</OrderDate>
</OrderForm>
```

For more examples of serialization, see [Examples of XML Serialization](#).

Items That Can Be Serialized

The following items can be serialized using the `XmlSerializer` class:

- Public read/write properties and fields of public classes.
- Classes that implement `ICollection` or `IEnumerable`.

 **Note**

Only collections are serialized, not public properties.

- `XmlElement` objects.
- `XmlNode` objects.
- `DataSet` objects.

For more information about serializing or deserializing objects, see [How to: Serialize an Object](#) and [How to: Deserialize an Object](#).

Advantages of Using XML Serialization

The **XmlSerializer** class gives you complete and flexible control when you serialize an object as XML. If you are creating an XML Web service, you can apply attributes that control serialization to classes and members to ensure that the XML output conforms to a specific schema.

For example, **XmlSerializer** enables you to:

- Specify whether a field or property should be encoded as an attribute or an element.
- Specify an XML namespace to use.
- Specify the name of an element or attribute if a field or property name is inappropriate.

Another advantage of XML serialization is that you have no constraints on the applications you develop, as long as the XML stream that is generated conforms to a given schema. Imagine a schema that is used to describe books. It features a title, author, publisher, and ISBN number element. You can develop an application that processes the XML data in any way you want, for example, as a book order, or as an inventory of books. In either case, the only requirement is that the XML stream conforms to the specified XML Schema definition language (XSD) schema.

XML Serialization Considerations

The following should be considered when using the **XmlSerializer** class:

- The Sgen.exe tool is expressly designed to generate serialization assemblies for optimum performance.
- The serialized data contains only the data itself and the structure of your classes. Type identity and assembly information are not included.
- Only public properties and fields can be serialized. Properties must have public accessors (get and set methods). If you must serialize non-public data, use the [DataContractSerializer](#) class rather than XML serialization.
- A class must have a parameterless constructor to be serialized by **XmlSerializer**.
- Methods cannot be serialized.
- **XmlSerializer** can process classes that implement **IEnumerable** or **ICollection** differently if they meet certain requirements, as follows.

A class that implements **IEnumerable** must implement a public **Add** method that takes a single parameter. The **Add** method's parameter must be consistent (polymorphic) with the type returned from the **IEnumerator.Current** property returned from the **GetEnumerator** method.

A class that implements **ICollection** in addition to **IEnumerable** (such as **CollectionBase**) must have a public **Item** indexed property (an indexer in C#) that takes an integer and it must have a public **Count** property of type **integer**. The parameter passed to the **Add** method must be the same type as that returned from the **Item** property, or one of that type's bases.

For classes that implement **ICollection**, values to be serialized are retrieved from the indexed **Item** property rather than by calling **GetEnumerator**. Also, public fields and properties are not serialized, with the exception of public fields that return another collection class (one that implements **ICollection**). For an example, see [Examples of XML Serialization](#).

XSD Data Type Mapping

The W3C document titled [XML Schema Part 2: Datatypes](#) specifies the simple data types that are allowed in an XML Schema definition language (XSD) schema. For many of these (for example, **int** and **decimal**), there is a corresponding data type in .NET. However, some XML data types do not have a corresponding .NET data type, for example, the **NMTOKEN** data type. In such cases, if you use the XML Schema Definition tool ([XML Schema Definition Tool \(Xsd.exe\)](#)) to generate classes from a schema, an appropriate attribute is applied to a member of type **string**, and its **DataType** property is set to the XML data type name. For example, if a schema contains an element named "MyToken" with the XML data type **NMTOKEN**, the generated class might contain a member as shown in the following example.

C#

```
[XmlElement(DataType = "NMTOKEN")]
public string MyToken;
```

Similarly, if you are creating a class that must conform to a specific XML Schema (XSD), you should apply the appropriate attribute and set its **DataType** property to the desired XML data type name.

For a complete list of type mappings, see the **DataType** property for any of the following attribute classes:

- [SoapAttributeAttribute](#)
- [SoapElementAttribute](#)
- [XmlAttributeAttribute](#)
- [XmlElementAttribute](#)
- [XmlRootAttribute](#)

See also

- [XmlSerializer](#)
- [DataContractSerializer](#)
- [FileStream](#)
- [XML and SOAP Serialization](#)
- [Binary Serialization](#)
- [Serialization](#)
- [XmlSerializer](#)
- [Examples of XML Serialization](#)
- [How to: Serialize an Object](#)
- [How to: Deserialize an Object](#)

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

Examples of XML Serialization

Article • 03/18/2023

XML serialization can take more than one form, from simple to complex. For example, you can serialize a class that simply consists of public fields and properties, as shown in [Introducing XML Serialization](#). The following code examples address various advanced scenarios, including how to use XML serialization to generate an XML stream that conforms to a specific XML Schema (XSD) document.

Serializing a DataSet

Besides serializing an instance of a public class, you can also serialize an instance of a [DataSet](#), as shown in the following code example:

C#

```
private void SerializeDataSet(string filename)
{
    XmlSerializer ser = new XmlSerializer(typeof(DataSet));

    // Creates a DataSet; adds a table, column, and ten rows.
    DataSet ds = new DataSet("myDataSet");
    DataTable t = new DataTable("table1");
    DataColumn c = new DataColumn("thing");
    t.Columns.Add(c);
    ds.Tables.Add(t);
    DataRow r;

    for (int i = 0; i < 10; i++) {
        r = t.NewRow();
        r[0] = "Thing " + i;
        t.Rows.Add(r);
    }

    TextWriter writer = new StreamWriter(filename);
    ser.Serialize(writer, ds);
    writer.Close();
}
```

Serializing an XElement and XmlNode

You can also serialize instances of an [XmlElement](#) or [XmlNode](#) class, as shown in the following code example:

C#

```

private void SerializeElement(string filename)
{
    XmlSerializer ser = new XmlSerializer(typeof(XmlElement));
    XmlElement myElement = new XmlDocument().CreateElement("MyElement",
    "ns");
    myElement.InnerText = "Hello World";
    TextWriter writer = new StreamWriter(filename);
    ser.Serialize(writer, myElement);
    writer.Close();
}

private void SerializeNode(string filename)
{
    XmlSerializer ser = new XmlSerializer(typeof(XmlNode));
    XmlNode myNode = new XmlDocument();
    CreateNode(XmlNodeType.Element, "MyNode", "ns");
    myNode.InnerText = "Hello Node";
    TextWriter writer = new StreamWriter(filename);
    ser.Serialize(writer, myNode);
    writer.Close();
}

```

Serializing a class that contains a field returning a complex object

If a property or field returns a complex object (such as an array or a class instance), the `XmlSerializer` converts it to an element nested within the main XML document. For example, the first class in the following code example returns an instance of the second class:

C#

```

public class PurchaseOrder
{
    public Address MyAddress;
}

public record Address
{
    public string FirstName;
}

```

The serialized XML output might look like this:

XML

```
<PurchaseOrder>
  <MyAddress>
    <FirstName>George</FirstName>
  </MyAddress>
</PurchaseOrder>
```

Serializing an array of objects

You can also serialize a field that returns an array of objects, as shown in the following code example:

```
C#
public class PurchaseOrder
{
    public Item [] ItemsOrders;
}

public class Item
{
    public string ItemID;
    public decimal ItemPrice;
}
```

If two items are ordered, the serialized class instance might look like the following code:

XML

```
<PurchaseOrder xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <ItemsOrders>
    <Item>
      <ItemID>aaa111</ItemID>
      <ItemPrice>34.22</ItemPrice>
    </Item>
    <Item>
      <ItemID>bbb222</ItemID>
      <ItemPrice>2.89</ItemPrice>
    </Item>
  </ItemsOrders>
</PurchaseOrder>
```

Serializing a class that implements the ICollection interface

You can create your own collection classes by implementing the [ICollection](#) interface and using the [XmlSerializer](#) to serialize instances of these classes.

① Note

When a class implements the [ICollection](#) interface, only the collection contained by the class is serialized. Any public properties or fields added to the class won't be serialized. To be serialized, the class must include an **Add** method and an **Item** property (C# indexer).

C#

```
using System;
using System.Collections;
using System.IO;
using System.Xml.Serialization;

public class Test
{
    static void Main()
    {
        Test t = new Test();
        t.SerializeCollection("coll.xml");
    }

    private void SerializeCollection(string filename)
    {
        Employees Emps = new Employees();
        // Note that only the collection is serialized -- not the
        // CollectionName or any other public property of the class.
        Emps.CollectionName = "Employees";
        Employee John100 = new Employee("John", "100xxx");
        Emps.Add(John100);
        XmlSerializer x = new XmlSerializer(typeof(Employees));
        TextWriter writer = new StreamWriter(filename);
        x.Serialize(writer, Emps);
    }
}

public class Employees : ICollection
{
    public string CollectionName;
    private ArrayList empArray = new ArrayList();

    public Employee this[int index] => (Employee) empArray[index];

    public void CopyTo(Array a, int index)
    {
        empArray.CopyTo(a, index);
    }
}
```

```

public int Count => empArray.Count;

public object SyncRoot => this;

public bool IsSynchronized => false;

public IEnumerator GetEnumerator() => empArray.GetEnumerator();

public void Add(Employee newEmployee)
{
    empArray.Add(newEmployee);
}
}

public class Employee
{
    public string EmpName;
    public string EmpID;

    public Employee() {}

    public Employee(string empName, string empID)
    {
        EmpName = empName;
        EmpID = empID;
    }
}

```

Purchase Order example

You can cut and paste the following example code into a text file and rename it with a .cs or .vb file name extension. Use the C# or Visual Basic compiler to compile the file. Then run it using the name of the executable.

This example uses a simple scenario to demonstrate how an instance of an object is created and serialized into a file stream using the [Serialize](#) method. The XML stream is saved to a file. The same file is then read and reconstructed into a copy of the original object using the [Deserialize](#) method.

In this example, a class named `PurchaseOrder` is serialized and then deserialized. A second class named `Address` is also included because the public field named `ShipTo` must be set to an `Address`. Similarly, an `OrderedItem` class is included because an array of `OrderedItem` objects must be set to the `OrderedItems` field. Finally, a class named `Test` contains the code that serializes and deserializes the classes.

The `CreatePO` method creates the `PurchaseOrder`, `Address`, and `OrderedItem` class objects and sets the public field values. The method also constructs an instance of the `XmlSerializer` class that's used to serialize and deserialize the `PurchaseOrder`.

ⓘ Note

The code passes the type of class that will be serialized to the constructor. The code also creates a `FileStream` that's used to write the XML stream to an XML document.

The `ReadPo` method is a little simpler. It just creates objects to deserialize and reads out their values. As with the `CreatePo` method, you must first construct an `XmlSerializer`, passing the type of class to be deserialized to the constructor. Also, a `FileStream` is required to read the XML document. To deserialize the objects, call the `Deserialize` method with the `FileStream` as an argument. The deserialized object must be cast to an object variable of type `PurchaseOrder`. The code then reads the values of the deserialized `PurchaseOrder`.

ⓘ Note

You can read the `PO.xml` file that's created to see the actual XML output.

C#

```
using System;
using System.IO;
using System.Xml;
using System.Xml.Serialization;

// The XmlRoot attribute allows you to set an alternate name
// (PurchaseOrder) for the XML element and its namespace. By
// default, the XmlSerializer uses the class name. The attribute
// also allows you to set the XML namespace for the element. Lastly,
// the attribute sets the IsNullable property, which specifies whether
// the xsi:null attribute appears if the class instance is set to
// a null reference.
[XmlRoot("PurchaseOrder", Namespace="http://www.cpandl.com",
IsNullable = false)]
public class PurchaseOrder
{
    public Address ShipTo;
    public string OrderDate;
    // The XmlArray attribute changes the XML element name
    // from the default of "OrderedItems" to "Items".
    [XmlArray("Items")]
}
```

```
    public OrderedItem[] OrderedItems;
    public decimal SubTotal;
    public decimal ShipCost;
    public decimal TotalCost;
}

public class Address
{
    // The XmlAttribute attribute instructs the XmlSerializer to serialize
    // the
    // Name field as an XML attribute instead of an XML element (XML element
    // is
    // the default behavior).
    [XmlAttribute]
    public string Name;
    public string Line1;

    // Setting the IsNullable property to false instructs the
    // XmlSerializer that the XML attribute will not appear if
    // the City field is set to a null reference.
    [XmlElement(IsNullable = false)]
    public string City;
    public string State;
    public string Zip;
}

public class OrderedItem
{
    public string ItemName;
    public string Description;
    public decimal UnitPrice;
    public int Quantity;
    public decimal LineTotal;

    // Calculate is a custom method that calculates the price per item
    // and stores the value in a field.
    public void Calculate()
    {
        LineTotal = UnitPrice * Quantity;
    }
}

public class Test
{
    public static void Main()
    {
        // Read and write purchase orders.
        Test t = new Test();
        t.CreatePO("po.xml");
        t.ReadPO("po.xml");
    }

    private void CreatePO(string filename)
    {
        // Creates an instance of the XmlSerializer class;
    }
}
```

```

// specifies the type of object to serialize.
XmlSerializer serializer = new XmlSerializer(typeof(PurchaseOrder));
TextWriter writer = new StreamWriter(filename);
PurchaseOrder po =new PurchaseOrder();

// Creates an address to ship and bill to.
Address billAddress = new Address();
billAddress.Name = "Teresa Atkinson";
billAddress.Line1 = "1 Main St.";
billAddress.City = "AnyTown";
billAddress.State = "WA";
billAddress.Zip = "00000";
// Sets ShipTo and BillTo to the same addressee.
po.ShipTo = billAddress;
po.OrderDate = System.DateTime.Now.ToString("yyyy-MM-dd");

// Creates an OrderedItem.
OrderedItem i1 = new OrderedItem();
i1.ItemName = "Widget S";
i1.Description = "Small widget";
i1.UnitPrice = (decimal) 5.23;
i1.Quantity = 3;
i1.Calculate();

// Inserts the item into the array.
OrderedItem [] items = {i1};
po.OrderedItems = items;
// Calculate the total cost.
decimal subTotal = new decimal();
foreach(OrderedItem oi in items)
{
    subTotal += oi.LineTotal;
}
po.SubTotal = subTotal;
po.ShipCost = (decimal) 12.51;
po.TotalCost = po.SubTotal + po.ShipCost;
// Serializes the purchase order, and closes the TextWriter.
serializer.Serialize(writer, po);
writer.Close();
}

protected void ReadPO(string filename)
{
    // Creates an instance of the XmlSerializer class;
    // specifies the type of object to be deserialized.
    XmlSerializer serializer = new XmlSerializer(typeof(PurchaseOrder));
    // If the XML document has been altered with unknown
    // nodes or attributes, handles them with the
    // UnknownNode and UnknownAttribute events.
    serializer.UnknownNode+= new
    XmlNodeEventHandler(serializer_UnknownNode);
    serializer.UnknownAttribute+= new
    XmlAttributeEventHandler(serializer_UnknownAttribute);

    // A FileStream is needed to read the XML document.
}

```

```

FileStream fs = new FileStream(filename, FileMode.Open);
// Declares an object variable of the type to be deserialized.
PurchaseOrder po;
// Uses the Deserialize method to restore the object's state
// with data from the XML document. */
po = (PurchaseOrder) serializer.Deserialize(fs);
// Reads the order date.
Console.WriteLine ("OrderDate: " + po.OrderDate);

// Reads the shipping address.
Address shipTo = po.ShipTo;
ReadAddress(shipTo, "Ship To:");
// Reads the list of ordered items.
OrderedItem [] items = po.OrderedItems;
Console.WriteLine("Items to be shipped:");
foreach(OrderedItem oi in items)
{
    Console.WriteLine("\t" +
        oi.ItemName + "\t" +
        oi.Description + "\t" +
        oi.UnitPrice + "\t" +
        oi.Quantity + "\t" +
        oi.LineTotal);
}
// Reads the subtotal, shipping cost, and total cost.
Console.WriteLine(
"\n\t\t\t\t Subtotal\t" + po.SubTotal +
"\n\t\t\t\t Shipping\t" + po.ShipCost +
"\n\t\t\t\t Total\t\t" + po.TotalCost
);
}

protected void ReadAddress(Address a, string label)
{
    // Reads the fields of the Address.
    Console.WriteLine(label);
    Console.Write("\t" +
        a.Name +"\n\t" +
        a.Line1 +"\n\t" +
        a.City +"\n\t" +
        a.State +"\n\t" +
        a.Zip +"\n");
}

protected void serializer_UnknownNode
(object sender, XmlNodeEventArgs e)
{
    Console.WriteLine("Unknown Node:" + e.Name + "\t" + e.Text);
}

protected void serializer_UnknownAttribute
(object sender, XmlAttributeEventArgs e)
{
    System.Xml.XmlAttribute attr = e.Attr;
    Console.WriteLine("Unknown attribute " +

```

```
        attr.Name + "=" + attr.Value + "'");
    }
}
```

The XML output might look like this:

XML

```
<?xml version="1.0" encoding="utf-8"?>
<PurchaseOrder xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns="http://www.cpandl.com">
  <ShipTo Name="Teresa Atkinson">
    <Line1>1 Main St.</Line1>
    <City>AnyTown</City>
    <State>WA</State>
    <Zip>00000</Zip>
  </ShipTo>
  <OrderDate>Wednesday, June 27, 2001</OrderDate>
  <Items>
    <OrderedItem>
      <ItemName>Widget S</ItemName>
      <Description>Small widget</Description>
      <UnitPrice>5.23</UnitPrice>
      <Quantity>3</Quantity>
      <LineTotal>15.69</LineTotal>
    </OrderedItem>
  </Items>
  <SubTotal>15.69</SubTotal>
  <ShipCost>12.51</ShipCost>
  <TotalCost>28.2</TotalCost>
</PurchaseOrder>
```

See also

- [Introducing XML serialization](#)
- [Controlling XML serialization using attributes](#)
- [Attributes that control XML serialization](#)
- [XmlSerializer class](#)
- [How to: Serialize an object](#)
- [How to: Deserialize an object](#)

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

 [Open a documentation issue](#)

 [Provide product feedback](#)

The XML Schema Definition Tool and XML Serialization

Article • 09/15/2021

The XML Schema Definition tool ([XML Schema Definition Tool \(Xsd.exe\)](#)) is installed along with the .NET Framework tools as part of the Windows® Software Development Kit (SDK). The tool is designed primarily for two purposes:

- To generate either C# or Visual Basic class files that conform to a specific XML Schema definition language (XSD) schema. The tool takes an XML Schema as an argument and outputs a file that contains a number of classes that, when serialized with the [XmlSerializer](#), conform to the schema. For information about how to use the tool to generate classes that conform to a specific schema, see [How to: Use the XML Schema Definition Tool to Generate Classes and XML Schema Documents](#).
- To generate an XML Schema document from a .dll file or .exe file. To see the schema of a set of files that you have either created or one that has been modified with attributes, pass the DLL or EXE as an argument to the tool to generate the XML schema. For information about how to use the tool to generate an XML Schema Document from a set of classes, see [How to: Use the XML Schema Definition Tool to Generate Classes and XML Schema Documents](#).

For more information about using the tool, see [XML Schema Definition Tool \(Xsd.exe\)](#).

See also

- [DataSet](#)
- [Introducing XML Serialization](#)
- [XML Schema Definition Tool \(Xsd.exe\)](#)
- [XmlSerializer](#)
- [How to: Serialize an Object](#)
- [How to: Deserialize an Object](#)
- [How to: Use the XML Schema Definition Tool to Generate Classes and XML Schema Documents](#)
- [XML Schema Binding Support](#)



Collaborate with us on



.NET feedback

GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

Control XML serialization using attributes

Article • 10/04/2022

Attributes can be used to control the XML serialization of an object or to create an alternate XML stream from the same set of classes. For more information about creating an alternate XML stream, see [How to: Specify an Alternate Element Name for an XML Stream](#).

ⓘ Note

If the XML generated must conform to section 5 of the World Wide Web Consortium (W3C) document titled [Simple Object Access Protocol \(SOAP\) 1.1](#), use the attributes listed in [Attributes That Control Encoded SOAP Serialization](#).

By default, an XML element name is determined by the class or member name. In a class named `Book`, a field named `ISBN` will produce an XML element tag `<ISBN>`, as shown in the following example:

C#

```
public class Book
{
    public string ISBN;
}
// When an instance of the Book class is serialized, it might
// produce this XML:
// <ISBN>1234567890</ISBN>.
```

The default behavior can be changed if you want to give the element a new name. The following code shows how an attribute enables this functionality by setting the `ElementName` property of an `XmlElementAttribute`:

C#

```
public class TaxRates {
    [XmlElement(ElementName = "TaxRate")]
    public decimal ReturnTaxRate;
}
```

For more information about attributes, see [Attributes](#). For a list of attributes that control XML serialization, see [Attributes That Control XML Serialization](#).

Controlling Array Serialization

The [XmlAttribute](#) and the [XmlAttributeAttribute](#) attributes control the serialization of arrays. Using these attributes, you can control the element name, namespace, and XML Schema (XSD) data type as defined in the W3C document titled [XML Schema Part 2: Datatypes](#). You can also specify the types that can be included in an array.

The [XmlAttribute](#) will determine the properties of the enclosing XML element that results when an array is serialized. For example, by default, serializing the array below will result in an XML element named `Employees`. The `Employees` element will contain a series of elements named after the array type `Employee`.

C#

```
public class Group {
    public Employee[] Employees;
}
public class Employee {
    public string Name;
}
```

A serialized instance might resemble the following code:

XML

```
<Group>
<Employees>
    <Employee>
        <Name>Haley</Name>
    </Employee>
</Employees>
</Group>
```

By applying a [XmlAttribute](#), you can change the name of the XML element, as follows:

C#

```
public class Group {
    [XmlAttribute("TeamMembers")]
    public Employee[] Employees;
}
```

The resulting XML might resemble the following code:

XML

```
<Group>
<TeamMembers>
    <Employee>
        <Name>Haley</Name>
    </Employee>
</TeamMembers>
</Group>
```

The [XmlAttributeAttribute](#), on the other hand, controls how the items contained in the array are serialized.

ⓘ Note

The attribute is applied to the field returning the array.

C#

```
public class Group {
    [XmlAttribute("MemberName")]
    public Employee[] Employees;
}
```

The resulting XML might resemble the following code:

XML

```
<Group>
<Employees>
    <MemberName>Haley</MemberName>
</Employees>
</Group>
```

Serializing Derived Classes

Another use of the [XmlAttributeAttribute](#) is to allow the serialization of derived classes. For example, another class named `Manager` that derives from `Employee` can be added to the previous example. If you don't apply the [XmlAttributeAttribute](#), the code will fail at run time because the derived class type won't be recognized. To remedy this outcome, apply the attribute twice, each time setting the [Type](#) property for each acceptable type (base and derived).

C#

```
public class Group {
    [XmlElement(Type = typeof(Employee)),
     XmlElement(Type = typeof(Manager))]
    public Employee[] Employees;
}

public class Employee {
    public string Name;
}

public class Manager:Employee {
    public int Level;
}
```

A serialized instance might resemble the following code:

XML

```
<Group>
<Employees>
    <Employee>
        <Name>Haley</Name>
    </Employee>
    <Employee xsi:type = "Manager">
        <Name>Ann</Name>
        <Level>3</Level>
    </Employee>
</Employees>
</Group>
```

Serializing an Array as a Sequence of Elements

You can also serialize an array as a flat sequence of XML elements by applying a [XmlElementAttribute](#) to the field returning the array as follows:

C#

```
public class Group {
    [XmlElement]
    public Employee[] Employees;
}
```

A serialized instance might resemble the following code:

XML

```
<Group>
<Employees>
    <Name>Haley</Name>
```

```
</Employees>
<Employees>
    <Name>Noriko</Name>
</Employees>
<Employees>
    <Name>Marco</Name>
</Employees>
</Group>
```

Another way to differentiate the two XML streams is to use the XML Schema Definition tool to generate the XML Schema (XSD) document files from the compiled code. For more information on using the tool, see [The XML Schema Definition Tool and XML Serialization](#). When no attribute is applied to the field, the schema describes the element in the following manner:

XML

```
<xss:element minOccurs="0" maxOccurs = "1" name="Employees"
type="ArrayOfEmployee" />
```

When the [XmlElementAttribute](#) is applied to the field, the resulting schema describes the element as follows:

XML

```
<xss:element minOccurs="0" maxOccurs="unbounded" name="Employees"
type="Employee" />
```

Serializing an ArrayList

The [ArrayList](#) class can contain a collection of diverse objects. You can therefore use an [ArrayList](#) much as you use an array. Instead of creating a field that returns an array of typed objects, however, you can create a field that returns a single [ArrayList](#). However, as with arrays, you must inform the [XmlSerializer](#) of the types of objects the [ArrayList](#) contains. To accomplish this, assign multiple instances of the [XmlElementAttribute](#) to the field, as shown in the following example.

C#

```
public class Group {
    [XmlElement(Type = typeof(Employee)),
     XmlElement(Type = typeof(Manager))]
    public ArrayList Info;
}
```

Controlling Serialization of Classes Using XmlRootAttribute and XmlTypeAttribute

You can apply two attributes to a class only: [XmlRootAttribute](#) and [XmlAttribute](#).

These attributes are similar. The [XmlRootAttribute](#) can be applied to only one class: the class that, when serialized, represents the XML document's opening and closing element—in other words, the root element. The [XmlAttribute](#), on the other hand, can be applied to any class, including the root class.

For example, in the previous examples, the `Group` class is the root class, and all its public fields and properties become the XML elements found in the XML document. Therefore, you can have only one root class. By applying the [XmlRootAttribute](#), you can control the XML stream generated by the [XmlSerializer](#). For example, you can change the element name and namespace.

The [XmlAttribute](#) allows you to control the schema of the generated XML. This capability is useful when you need to publish the schema through an XML Web service. The following example applies both the [XmlAttribute](#) and the [XmlRootAttribute](#) to the same class:

```
C#  
  
[XmlRoot("NewGroupName")]
[XmlAttribute("NewTypeName")]
public class Group {
    public Employee[] Employees;
}
```

If this class is compiled, and the XML Schema Definition tool is used to generate its schema, you would find the following XML describing `Group`:

```
XML  
  
<xss:element name="NewGroupName" type="NewTypeName" />
```

In contrast, if you were to serialize an instance of the class, only `NewGroupName` would be found in the XML document:

```
XML  
  
<NewGroupName>
    ...
</NewGroupName>
```

Preventing Serialization with the `XmllgnoreAttribute`

You might come across a situation where a public property or field doesn't need to be serialized. For example, a field or property could be used to contain metadata. In such cases, apply the [XmllgnoreAttribute](#) to the field or property and the [XmlSerializer](#) will skip over it.

See also

- [Attributes That Control XML Serialization](#)
- [Attributes That Control Encoded SOAP Serialization](#)
- [Introducing XML Serialization](#)
- [Examples of XML Serialization](#)
- [How to: Specify an Alternate Element Name for an XML Stream](#)
- [How to: Serialize an Object](#)
- [How to: Deserialize an Object](#)

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback [here](#).

 [Open a documentation issue](#)

 [Provide product feedback](#)

Attributes That Control XML Serialization

Article • 09/15/2021

You can apply the attributes in the following table to classes and class members to control the way in which the [XmlSerializer](#) serializes or deserializes an instance of the class. To understand how these attributes control XML serialization, see [Controlling XML Serialization Using Attributes](#).

These attributes can also be used to control the literal style SOAP messages generated by an XML Web service. For more information about applying these attributes to an XML Web services method, see [XML Serialization with XML Web Services](#).

For more information about attributes, see [Attributes](#).

| Attribute | Applies to | Specifies |
|---------------------------------------|---|---|
| XmlAttributeAttribute | Public field, property, parameter, or return value that returns an array of XmlAttribute objects. | When deserializing, the array will be filled with XmlAttribute objects that represent all XML attributes unknown to the schema. |
| XmlElementAttribute | Public field, property, parameter, or return value that returns an array of XmlElement objects. | When deserializing, the array is filled with XmlElement objects that represent all XML elements unknown to the schema. |
| XmlAttributeAttribute | Public field, property, parameter, or return value that returns an array of complex objects. | The members of the array will be generated as members of an XML array. |
| XmlElementAttribute | Public field, property, parameter, or return value that returns an array of complex objects. | The derived types that can be inserted into an array. Usually applied in conjunction with an XmlAttributeAttribute . |
| XmlAttributeAttribute | Public field, property, parameter, or return value. | The member will be serialized as an XML attribute. |
| XmlElementAttribute | Public field, property, parameter, or return value. | The member can be further disambiguated by using an enumeration. |
| XmlElementAttribute | Public field, property, parameter, or return value. | The field or property will be serialized as an XML element. |

| Attribute | Applies to | Specifies |
|------------------------------|--|---|
| XmlAttribute | Public field that is an enumeration identifier. | The element name of an enumeration member. |
| XmlAttribute | Public properties and fields. | The property or field should be ignored when the containing class is serialized. |
| XmlAttribute | Public derived class declarations, and return values of public methods for Web Services Description Language (WSDL) documents. | The class should be included when generating schemas (to be recognized when serialized). |
| XmlAttribute | Public class declarations. | Controls XML serialization of the attribute target as an XML root element. Use the attribute to further specify the namespace and element name. |
| XmlAttribute | Public properties and fields. | The property or field should be serialized as XML text. |
| XmlAttribute | Public class declarations. | The name and namespace of the XML type. |

In addition to these attributes, which are all found in the [System.Xml.Serialization](#) namespace, you can also apply the [DefaultValueAttribute](#) attribute to a field. The [DefaultValueAttribute](#) sets the value that will be automatically assigned to the member if no value is specified.

To control encoded SOAP XML serialization, see [Attributes That Control Encoded SOAP Serialization](#).

See also

- [XML and SOAP Serialization](#)
- [XmlSerializer](#)
- [Controlling XML Serialization Using Attributes](#)
- [How to: Specify an Alternate Element Name for an XML Stream](#)
- [How to: Serialize an Object](#)
- [How to: Deserialize an Object](#)

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

XML Serialization with XML Web Services

Article • 03/30/2023

XML serialization is the underlying transport mechanism used in the XML Web services architecture, performed by the [XmlSerializer](#) class. To control the XML generated by an XML Web service, you can apply the attributes listed in both [Attributes That Control XML Serialization](#) and [Attributes That Control Encoded SOAP Serialization](#) to the classes, return values, parameters, and fields of a file used to create an XML Web service (.asmx). For more information about creating an XML Web service, see [XML Web Services Using ASP.NET](#).

Literal and Encoded Styles

The XML generated by an XML Web service can be formatted in either one of two ways, either literal or encoded, as explained in [Customizing SOAP Message Formatting](#).

Therefore there are two sets of attributes that control XML serialization. The attributes listed in [Attributes That Control XML Serialization](#) are designed to control literal style XML. The attributes listed in [Attributes That Control Encoded SOAP Serialization](#) control the encoded style. By selectively applying these attributes, you can tailor an application to return either, or both styles. Furthermore, these attributes can be applied (as appropriate) to return values and parameters.

Example of Using Both Styles

When you're creating an XML Web service, you can use both sets of attributes on the methods. In the following code example, the class named `MyService` contains two XML Web service methods, `MyLiteralMethod` and `MyEncodedMethod`. Both methods perform the same function: returning an instance of the `Order` class. In the `Order` class, the [XmlAttribute](#) and the [SoapTypeAttribute](#) attributes are both applied to the `OrderID` field, and both attributes have their `ElementName` property set to different values.

To run the example, paste the code into a file with an .asmx extension, and place the file into a virtual directory managed by Internet Information Services (IIS). From a web browser, type the name of the computer, virtual directory, and file.

C#

```

<%@ WebService Language="C#" Class="MyService" %>
using System;
using System.Web.Services;
using System.Web.Services.Protocols;
using System.Xml.Serialization;
public class Order {
    // Both types of attributes can be applied. Depending on which type
    // the method used, either one will affect the call.
    [SoapElement(ElementName = "EncodedOrderID")]
    [XmlElement(ElementName = "LiteralOrderID")]
    public String OrderID;
}
public class MyService {
    [WebMethod][SoapDocumentMethod]
    public Order MyLiteralMethod(){
        Order myOrder = new Order();
        return myOrder;
    }
    [WebMethod][SoapRpcMethod]
    public Order MyEncodedMethod(){
        Order myOrder = new Order();
        return myOrder;
    }
}

```

The following code example calls `MyLiteralMethod`. The element name is changed to "LiteralOrderID".

XML

```

<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
    <soap:Body>
        <MyLiteralMethodResponse xmlns="http://tempuri.org/">
            <MyLiteralMethodResult>
                <LiteralOrderID>string</LiteralOrderID>
            </MyLiteralMethodResult>
        </MyLiteralMethodResponse>
    </soap:Body>
</soap:Envelope>

```

The following code example calls `MyEncodedMethod`. The element name is "EncodedOrderID".

XML

```

<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
    <soap:Body>
        <MyEncodedMethodResponse xmlns="http://tempuri.org/">
            <MyEncodedMethodResult>
                <EncodedOrderID>string</EncodedOrderID>
            </MyEncodedMethodResult>
        </MyEncodedMethodResponse>
    </soap:Body>
</soap:Envelope>

```

```

xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:tns="http://tempuri.org/"
xmlns:types="http://tempuri.org/encodedTypes"
xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
    <soap:Body>
        soap:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
            <tns:MyEncodedMethodResponse>
                <MyEncodedMethodResult href="#id1" />
            </tns:MyEncodedMethodResponse>
            <types:Order id="id1" xsi:type="types:Order">
                <EncodedOrderID xsi:type="xsd:string">string</EncodedOrderID>
            </types:Order>
        </soap:Body>
    </soap:Envelope>

```

Applying Attributes to Return Values

You can also apply attributes to return values to control the namespace, element name, and so forth. The following code example applies the `XmlElementAttribute` attribute to the return value of the `MyLiteralMethod` method. Doing so allows you to control the namespace and element name.

C#

```

[return: XmlElement(Namespace = "http://www.cohowinery.com",
    ElementName = "BookOrder")]
[WebMethod][SoapDocumentMethod]
public Order MyLiteralMethod(){
    Order myOrder = new Order();
    return myOrder;
}

```

When invoked, the code returns XML that resembles the following.

XML

```

<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
    <soap:Body>
        <MyLiteralMethodResponse xmlns="http://tempuri.org/">
            <BookOrder xmlns="http://www.cohowinery.com">
                <LiteralOrderID>string</LiteralOrderID>
            </BookOrder>
        </MyLiteralMethodResponse>
    </soap:Body>

```

```
</soap:Body>
</soap:Envelope>
```

Attributes Applied to Parameters

You can also apply attributes to parameters to specify namespace, element name and so forth. The following code example adds a parameter to the `MyLiteralMethodResponse` method, and applies the `XmlAttributeAttribute` attribute to the parameter. The element name and namespace are both set for the parameter.

C#

```
[return: XmlElement(Namespace = "http://www.cohowinery.com",
ElementName = "BookOrder")]
[WebMethod][SoapDocumentMethod]
public Order MyLiteralMethod([XmlElement("MyOrderID",
Namespace="http://www.microsoft.com")] string ID){
    Order myOrder = new Order();
    myOrder.OrderID = ID;
    return myOrder;
}
```

The SOAP request would resemble the following.

XML

```
<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
    <soap:Body>
        <MyLiteralMethod xmlns="http://tempuri.org/">
            <MyOrderID xmlns="http://www.microsoft.com">string</MyOrderID>
        </MyLiteralMethod>
    </soap:Body>
</soap:Envelope>
```

Applying Attributes to Classes

If you need to control the namespace of elements that correlate to classes, you can apply `XmlAttributeAttribute`, `XmlRootAttribute`, and `SoapTypeAttribute`, as appropriate. The following code example applies all three to the `Order` class.

C#

```
[XmlAttribute("BigBooksService", Namespace = "http://www.cpandl.com")]
[SoapType("SoapBookService")]
[XmlRoot("BookOrderForm")]
public class Order {
    // Both types of attributes can be applied. Depending on which
    // the method used, either one will affect the call.
    [SoapElement(ElementName = "EncodedOrderID")]
    [XmlElement(ElementName = "LiteralOrderID")]
    public String OrderID;
}
```

The results of applying the `XmlAttribute` and `SoapTypeAttribute` can be seen when you examine the service description, as shown in the following code example.

XML

```
<s:element name="BookOrderForm" type="s0:BigBookService" />
<s:complexType name="BigBookService">
    <s:sequence>
        <s:element minOccurs="0" maxOccurs="1" name="LiteralOrderID"
type="s:string" />
    </s:sequence>

    <s:schema targetNamespace="http://tempuri.org/encodedTypes">
        <s:complexType name="SoapBookService">
            <s:sequence>
                <s:element minOccurs="1" maxOccurs="1" name="EncodedOrderID"
type="s:string" />
            </s:sequence>
        </s:complexType>
    </s:schema>
</s:complexType>
```

The effect of the `XmlAttribute` can also be seen in the HTTP GET and HTTP POST results, as follows.

XML

```
<?xml version="1.0" encoding="utf-8"?>
<BookOrderForm xmlns="http://tempuri.org/">
    <LiteralOrderID>string</LiteralOrderID>
</BookOrderForm>
```

See also

- [XML and SOAP Serialization](#)
- [Attributes That Control Encoded SOAP Serialization](#)

- How to: Serialize an Object as a SOAP-Encoded XML Stream
- How to: Override Encoded SOAP XML Serialization
- Introducing XML Serialization
- How to: Serialize an Object
- How to: Deserialize an Object

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

Attributes That Control Encoded SOAP Serialization

Article • 09/15/2021

The World Wide Web Consortium (W3C) document named [Simple Object Access Protocol \(SOAP\) 1.1](#) contains an optional section (section 5) that describes how SOAP parameters can be encoded. To conform to section 5 of the specification, you must use a special set of attributes found in the [System.Xml.Serialization](#) namespace. Apply those attributes as appropriate to classes and members of classes, and then use the [XmlSerializer](#) to serialize instances of the class or classes.

The following table shows the attributes, where they can be applied, and what they do. For more information about using these attributes to control XML serialization, see [How to: Serialize an Object as a SOAP-Encoded XML Stream](#) and [How to: Override Encoded SOAP XML Serialization](#).

For more information about attributes, see [Attributes](#).

| Attribute | Applies to | Specifies |
|--|--|---|
| SoapAttributeAttribute | Public field, property, parameter, or return value. | The class member will be serialized as an XML attribute. |
| SoapElementAttribute | Public field, property, parameter, or return value. | The class will be serialized as an XML element. |
| SoapEnumAttribute | Public field that is an enumeration identifier. | The element name of an enumeration member. |
| SoapIgnoreAttribute | Public properties and fields. | The property or field should be ignored when the containing class is serialized. |
| SoapIncludeAttribute | Public-derived class declarations and public methods for Web Services Description Language (WSDL) documents. | The type should be included when generating schemas (to be recognized when serialized). |
| SoapTypeAttribute | Public class declarations. | The class should be serialized as an XML type. |

See also

- [XML and SOAP Serialization](#)

- [How to: Serialize an Object as a SOAP-Encoded XML Stream](#)
- [How to: Override Encoded SOAP XML Serialization](#)
- [Attributes](#)
- [XmlSerializer](#)
- [How to: Serialize an Object](#)
- [How to: Deserialize an Object](#)

 [Collaborate with us on GitHub](#)

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

How to: Serialize an Object

Article • 09/15/2021

To serialize an object, first create the object that is to be serialized and set its public properties and fields. To do this, you must determine the transport format in which the XML stream is to be stored, either as a stream or as a file. For example, if the XML stream must be saved in a permanent form, create a [FileStream](#) object.

ⓘ Note

For more examples of XML serialization, see [Examples of XML Serialization](#).

To serialize an object

1. Create the object and set its public fields and properties.
2. Construct a [XmlSerializer](#) using the type of the object. For more information, see the [XmlSerializer](#) class constructors.
3. Call the [Serialize](#) method to generate either an XML stream or a file representation of the object's public properties and fields. The following example creates a file.

C#

```
MySerializableClass myObject = new MySerializableClass();
// Insert code to set properties and fields of the object.
XmlSerializer mySerializer = new
XmlSerializer(typeof(MySerializableClass));
// To write to a file, create a StreamWriter object.
StreamWriter myWriter = new StreamWriter("myFileName.xml");
mySerializer.Serialize(myWriter, myObject);
myWriter.Close();
```

See also

- [Introducing XML Serialization](#)
- [How to: Deserialize an Object](#)

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

How to deserialize an object using XmlSerializer

Article • 09/15/2021

When you deserialize an object, the transport format determines whether you will create a stream or file object. After the transport format is determined, you can call the [Serialize](#) or [Deserialize](#) methods, as required.

To deserialize an object

1. Construct a [XmlSerializer](#) using the type of the object to deserialize.
2. Call the [Deserialize](#) method to produce a replica of the object. When deserializing, you must cast the returned object to the type of the original, as shown in the following example, which deserializes the object from a file (although it could also be serialized from a stream).

C#

```
// Construct an instance of the XmlSerializer with the type
// of object that is being deserialized.
var mySerializer = new XmlSerializer(typeof(MySerializableClass));
// To read the file, create a FileStream.
using var myFileStream = new FileStream("myFileName.xml",
    FileMode.Open);
// Call the Deserialize method and cast to the object type.
var myObject =
    (MySerializableClass)mySerializer.Deserialize(myFileStream);
```

See also

- [Introducing XML Serialization](#)
- [How to: Serialize an Object](#)

 Collaborate with us on
GitHub

The source for this content can
be found on GitHub, where you
can also create and review

.NET

.NET feedback

The .NET documentation is open
source. Provide feedback [here](#).

issues and pull requests. For more information, see [our contributor guide](#).

 Open a documentation issue

 Provide product feedback

How to: Use the XML Schema Definition Tool to Generate Classes and XML Schema Documents

Article • 06/01/2023

The XML Schema Definition tool (Xsd.exe) allows you to generate an XML schema that describes a class or to generate the class defined by an XML schema. The following procedures show how to perform these operations.

The XML Schema Definition tool (Xsd.exe) usually can be found in the following path:
C:\Program Files (x86)\Microsoft SDKs\Windows\{version}\bin\NETFX {version} Tools\

To generate classes that conform to a specific schema

1. Open a command prompt.
2. Pass the XML Schema as an argument to the XML Schema Definition tool, which creates a set of classes that are precisely matched to the XML Schema, for example:

```
Console  
xsd mySchema.xsd
```

The tool can only process schemas that reference the World Wide Web Consortium XML specification of March 16, 2001. In other words, the XML Schema namespace must be "http://www.w3.org/2001/XMLSchema", as shown in the following example.

```
XML  
<?xml version="1.0" encoding="utf-8"?>  
<xss:schema attributeFormDefault="qualified"  
elementFormDefault="qualified" targetNamespace=""  
xmlns:xss="http://www.w3.org/2001/XMLSchema" />
```

3. Modify the classes with methods, properties, or fields, as necessary. For more information about modifying a class with attributes, see [Controlling XML](#)

[Serialization Using Attributes](#) and [Attributes That Control Encoded SOAP Serialization](#).

It is often useful to examine the schema of the XML stream that is generated when instances of a class (or classes) are serialized. For example, you might publish your schema for others to use, or you might compare it to a schema with which you are trying to achieve conformity.

To generate an XML Schema document from a set of classes

1. Compile the class or classes into a DLL.
2. Open a command prompt.
3. Pass the DLL as an argument to Xsd.exe, for example:

```
Console  
xsd MyFile.dll
```

The schema (or schemas) will be written, beginning with the name "schema0.xsd".

See also

- [DataSet](#)
- [The XML Schema Definition Tool and XML Serialization](#)
- [Introducing XML Serialization](#)
- [XML Schema Definition Tool \(Xsd.exe\)](#)
- [XmlSerializer](#)
- [How to: Serialize an Object](#)
- [How to: Deserialize an Object](#)

 Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

more information, see [our contributor guide](#).

 [Provide product feedback](#)

How to: Control Serialization of Derived Classes

Article • 09/15/2021

Using the **XmlElementAttribute** attribute to change the name of an XML element is not the only way to customize object serialization. You can also customize the XML stream by deriving from an existing class and instructing the **XmlSerializer** instance how to serialize the new class.

For example, given a **Book** class, you can derive from it and create an **ExpandedBook** class that has a few more properties. However, you must instruct the **XmlSerializer** to accept the derived type when serializing or deserializing. This can be done by creating a **XmlElementAttribute** instance and setting its **Type** property to the derived class type. Add the **XmlElementAttribute** to a **XmlAttribute** instance. Then add the **XmlAttribute** to a **XmlAttributeOverrides** instance, specifying the type being overridden and the name of the member that accepts the derived class. This is shown in the following example.

Example

C#

```
public class Orders
{
    public Book[] Books;
}

public class Book
{
    public string ISBN;
}

public class ExpandedBook:Book
{
    public bool NewEdition;
}

public class Run
{
    public void SerializeObject(string filename)
    {
        // Each overridden field, property, or type requires
        // an XmlAttributes instance.
        XmlAttributes attrs = new XmlAttributes();

        // Creates an XmlElementAttribute instance to override the
        // name of the Books element.
        attrs.ElementName = "MyBooks";
        attrs.Type = typeof(ExpandedBook);
        attrs.XmlElement = new XmlElementAttribute("MyBooks");
        attrs.XmlElement.Type = typeof(ExpandedBook);
    }
}
```

```

// field that returns Book objects. The overridden field
// returns Expanded objects instead.
XmlElementAttribute attr = new XmlElementAttribute();
attr.ElementName = "NewBook";
attr.Type = typeof(ExpandedBook);

// Adds the element to the collection of elements.
attrs.XmlElements.Add(attr);

// Creates the XmlAttributeOverrides instance.
XmlAttributeOverrides attrOverrides = new XmlAttributeOverrides();

// Adds the type of the class that contains the overridden
// member, as well as the XmlAttributes instance to override it
// with, to the XmlAttributeOverrides.
attrOverrides.Add(typeof(Orders), "Books", attrs);

// Creates the XmlSerializer using the XmlAttributeOverrides.
XmlSerializer s =
new XmlSerializer(typeof(Orders), attrOverrides);

// Writing the file requires a TextWriter instance.
TextWriter writer = new StreamWriter(filename);

// Creates the object to be serialized.
Orders myOrders = new Orders();

// Creates an object of the derived type.
ExpandedBook b = new ExpandedBook();
b.ISBN= "123456789";
b.NewEdition = true;
myOrders.Books = new ExpandedBook[] {b};

// Serializes the object.
s.Serialize(writer,myOrders);
writer.Close();
}

public void DeserializeObject(string filename)
{
    XmlAttributeOverrides attrOverrides =
        new XmlAttributeOverrides();
    XmlAttributes attrs = new XmlAttributes();

    // Creates an XmlElementAttribute to override the
    // field that returns Book objects. The overridden field
    // returns Expanded objects instead.
    XmlElementAttribute attr = new XmlElementAttribute();
    attr.ElementName = "NewBook";
    attr.Type = typeof(ExpandedBook);

    // Adds the XmlElementAttribute to the collection of objects.
    attrs.XmlElements.Add(attr);

    attrOverrides.Add(typeof(Orders), "Books", attrs);
}

```

```
// Creates the XmlSerializer using the XmlAttributeOverrides.
XmlSerializer s =
    new XmlSerializer(typeof(Orders), attrOverrides);

FileStream fs = new FileStream(filename, FileMode.Open);
Orders myOrders = (Orders) s.Deserialize(fs);
Console.WriteLine("ExpandedBook:");

// The difference between deserializing the overridden
// XML document and serializing it is this: To read the derived
// object values, you must declare an object of the derived type
// and cast the returned object to it.
ExpandedBook expanded;
foreach(Book b in myOrders.Books)
{
    expanded = (ExpandedBook)b;
    Console.WriteLine(
        expanded.ISBN + "\n" +
        expanded.NewEdition);
}
}
```

See also

- [XmlAttributeOverrides](#)
- [XmlElementAttribute](#)
- [XmlAttributeOverrides](#)
- [XmlAttributeOverrides](#)
- [XML and SOAP Serialization](#)
- [How to: Serialize an Object](#)
- [How to: Specify an Alternate Element Name for an XML Stream](#)

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

How to: Specify an Alternate Element Name for an XML Stream

Article • 09/15/2021

Using the [XmlSerializer](#), you can generate more than one XML stream with the same set of classes. You might want to do this because two different XML Web services require the same basic information, with only slight differences. For example, imagine two XML Web services that process orders for books, and thus both require ISBN numbers. One service uses the tag <ISBN> while the second uses the tag <BookID>. You have a class named `Book` that contains a field named `ISBN`. When an instance of the `Book` class is serialized, it will, by default, use the member name (`ISBN`) as the tag element name. For the first XML Web service, this is as expected. But to send the XML stream to the second XML Web service, you must override the serialization so that the tag's element name is `BookID`.

To create an XML stream with an alternate element name

1. Create an instance of the [XmlElementAttribute](#) class.
2. Set the [ElementName](#) of the [XmlElementAttribute](#) to "BookID".
3. Create an instance of the [XmlAttributeOverrides](#) class.
4. Add the `XmlElementAttribute` object to the collection accessed through the [XmlElements](#) property of [XmlAttributeOverrides](#) .
5. Create an instance of the [XmlAttributeOverrides](#) class.
6. Add the `XmlAttributeOverrides` to the [XmlAttributeOverrides](#), passing the type of the object to override and the name of the member being overridden.
7. Create an instance of the `XmlSerializer` class with `XmlAttributeOverrides`.
8. Create an instance of the `Book` class, and serialize or deserialize it.

Example

C#

```
public void SerializeOverride()
{
    // Creates an XmlElementAttribute with the alternate name.
    XmlElementAttribute myElementAttribute = new XmlElementAttribute();
    myElementAttribute.ElementName = "BookID";
    XmlAttributes myAttributes = new XmlAttributes();
    myAttributes.XmlElements.Add(myElementAttribute);
    XmlAttributeOverrides myOverrides = new XmlAttributeOverrides();
    myOverrides.Add(typeof(Book), "ISBN", myAttributes);
    XmlSerializer mySerializer =
        new XmlSerializer(typeof(Book), myOverrides);
    Book b = new Book();
    b.ISBN = "123456789";
    // Creates a StreamWriter to write the XML stream to.
    StreamWriter writer = new StreamWriter("Book.xml");
    mySerializer.Serialize(writer, b);
}
```

The XML stream might resemble the following.

XML

```
<Book>
    <BookID>123456789</BookID>
</Book>
```

See also

- [XmlElementAttribute](#)
- [XmlAttribute](#)
- [XmlAttributeOverrides](#)
- [XML and SOAP Serialization](#)
- [XmlSerializer](#)
- [How to: Serialize an Object](#)
- [How to: Deserialize an Object](#)

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 Open a documentation issue

more information, see [our contributor guide](#).

 [Provide product feedback](#)

How to qualify XML element and XML attribute names

Article • 09/15/2021

XML namespaces contained by instances of the [XmlSerializerNamespaces](#) class must conform to the World Wide Web Consortium (W3C) specification called [Namespaces in XML](#).

XML namespaces provide a method for qualifying the names of XML elements and XML attributes in XML documents. A qualified name consists of a prefix and a local name, separated by a colon. The prefix functions only as a placeholder; it is mapped to a URI that specifies a namespace. The combination of the universally managed URI namespace and the local name produces a name that is guaranteed to be universally unique.

By creating an instance of `XmlSerializerNamespaces` and adding the namespace pairs to the object, you can specify the prefixes used in an XML document.

To create qualified names in an XML document

1. Create an instance of the `XmlSerializerNamespaces` class.
2. Add all prefixes and namespace pairs to the `XmlSerializerNamespaces`.
3. Apply the appropriate `System.Xml.Serialization` attribute to each member or class that the [XmlSerializer](#) is to serialize into an XML document.
The available attributes are: [XmlElementAttribute](#), [XmlAttributeAttribute](#), [XmlAttributeItemAttribute](#), [XmlAttributeAttribute](#), [XmlElementAttribute](#), [XmlAttributeAttribute](#), and [XmlAttributeAttribute](#).
4. Set the `Namespace` property of each attribute to one of the namespace values from the `XmlSerializerNamespaces`.
5. Pass the `XmlSerializerNamespaces` to the `Serialize` method of the `XmlSerializer`.

Example

The following example creates an `XmlSerializerNamespaces`, and adds two prefix and namespace pairs to the object. The code creates an `XmlSerializer` that is used to

serialize an instance of the `Books` class. The code calls the `Serialize` method with the `XmlSerializerNamespaces`, allowing the XML to contain prefixed namespaces.

C#

```
using System;
using System.IO;
using System.Xml;
using System.Xml.Serialization;

public class Program
{
    public static void Main()
    {
        SerializeObject("XmlNamespaces.xml");
    }

    public static void SerializeObject(string filename)
    {
        var mySerializer = new XmlSerializer(typeof(Books));
        // Writing a file requires a TextWriter.
        TextWriter myWriter = new StreamWriter(filename);

        // Creates an XmlSerializerNamespaces and adds two
        // prefix-namespace pairs.
        var myNamespaces = new XmlSerializerNamespaces();
        myNamespaces.Add("books", "http://www.cpandl.com");
        myNamespaces.Add("money", "http://www.cohowinery.com");

        // Creates a Book.
        var myBook = new Book();
        myBook.TITLE = "A Book Title";
        var myPrice = new Price();
        myPrice.price = (decimal) 9.95;
        myPrice.currency = "US Dollar";
        myBook.PRICE = myPrice;
        var myBooks = new Books();
        myBooks.Book = myBook;
        mySerializer.Serialize(myWriter, myBooks, myNamespaces);
        myWriter.Close();
    }
}

public class Books
{
    [XmlElement(Namespace = "http://www.cohowinery.com")]
    public Book Book;
}

[XmlAttribute(Namespace = "http://www.cpandl.com")]
public class Book
{
    [XmlElement(Namespace = "http://www.cpandl.com")]
```

```
public string TITLE;
[XmlElement(Namespace = "http://www.cohowinery.com")]
public Price PRICE;
}

public class Price
{
    [XmlAttribute(Namespace = "http://www.cpndl.com")]
    public string currency;
    [XmlElement(Namespace = "http://www.cohowinery.com")]
    public decimal price;
}
```

See also

- [XmlSerializer](#)
- [The XML Schema Definition Tool and XML Serialization](#)
- [Introducing XML Serialization](#)
- [XmlSerializer Class](#)
- [Attributes That Control XML Serialization](#)
- [How to: Specify an Alternate Element Name for an XML Stream](#)
- [How to: Serialize an Object](#)
- [How to: Deserialize an Object](#)

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

How to: Serialize an Object as a SOAP-Encoded XML Stream

Article • 09/15/2021

Because a SOAP message is built using XML, the [XmlSerializer](#) class can be used to serialize classes and generate encoded SOAP messages. The resulting XML conforms to section 5 of the World Wide Web Consortium document "Simple Object Access Protocol (SOAP) 1.1" [↗](#). When you are creating an XML Web service that communicates through SOAP messages, you can customize the XML stream by applying a set of special SOAP attributes to classes and members of classes. For a list of attributes, see [Attributes That Control Encoded SOAP Serialization](#).

To serialize an object as a SOAP-encoded XML stream

1. Create the class using the [XML Schema Definition Tool \(Xsd.exe\)](#).
2. Apply one or more of the special attributes found in [System.Xml.Serialization](#).
See the list in "Attributes That Control Encoded SOAP Serialization."
3. Create an [XmlAttributeMapping](#) by creating a new [SoapReflectionImporter](#), and invoking the [ImportTypeMapping](#) method with the type of the serialized class.

The following code example calls the [ImportTypeMapping](#) method of the [SoapReflectionImporter](#) class to create an [XmlAttributeMapping](#).

C#

```
// Serializes a class named Group as a SOAP message.  
XmlAttributeMapping myTypeMapping =  
    new SoapReflectionImporter().ImportTypeMapping(typeof(Group));
```

4. Create an instance of the [XmlSerializer](#) class by passing the [XmlAttributeMapping](#) to the [XmlSerializer\(XmlTypeMapping\)](#) constructor.

C#

```
XmlAttributeMapper mySerializer = new XmlSerializer(myTypeMapping);
```

5. Call the [Serialize](#) or [Deserialize](#) method.

Example

C#

```
// Serializes a class named Group as a SOAP message.  
XmlTypeMapping myTypeMapping =  
    new SoapReflectionImporter().ImportTypeMapping(typeof(Group));  
XmlSerializer mySerializer = new XmlSerializer(myTypeMapping);
```

See also

- [XML and SOAP Serialization](#)
- [Attributes That Control Encoded SOAP Serialization](#)
- [XML Serialization with XML Web Services](#)
- [How to: Serialize an Object](#)
- [How to: Deserialize an Object](#)
- [How to: Override Encoded SOAP XML Serialization](#)

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

How to: Override Encoded SOAP XML Serialization

Article • 09/15/2021

The process for overriding XML serialization of objects as SOAP messages is similar to the process for overriding standard XML serialization. For information about overriding standard XML serialization, see [How to: Specify an Alternate Element Name for an XML Stream](#).

To override serialization of objects as SOAP messages

1. Create an instance of the `SoapAttributeOverrides` class.
2. Create a `SoapAttributes` for each class member that is being serialized.
3. Create an instance of one or more of the attributes that affect XML serialization, as appropriate, to the member being serialized. For more information, see "Attributes That Control Encoded SOAP Serialization".
4. Set the appropriate property of `SoapAttributes` to the attribute created in step 3.
5. Add `SoapAttributes` to `SoapAttributeOverrides`.
6. Create an `XmlTypeMapping` using the `SoapAttributeOverrides`. Use the `SoapReflectionImporter.ImportTypeMapping` method.
7. Create an `XmlSerializer` using `XmlTypeMapping`.
8. Serialize or deserialize the object.

Example

The following code example serializes a file in two ways: first, without overriding the `XmlSerializer` class's behavior, and second, by overriding the behavior. The example contains a class named `Group` with several members. Various attributes, such as the `SoapElementAttribute`, have been applied to class members. When the class is serialized with the `SerializeOriginal` method, the attributes control the SOAP message content. When the `SerializeOverride` method is called, the behavior of the `XmlSerializer` is

overridden by creating various attributes and setting the properties of a `SoapAttributes` to those attributes (as appropriate).

C#

```
using System;
using System.IO;
using System.Xml;
using System.Xml.Serialization;
using System.Xml.Schema;

public class Group
{
    [SoapAttribute(Namespace = "http://www.cpandl.com")]
    public string GroupName;

    [SoapAttribute(DataType = "base64Binary")]
    public Byte [] GroupNumber;

    [SoapAttribute(DataType = "date", AttributeName = "CreationDate")]
    public DateTime Today;
    [SoapElement(DataType = "nonNegativeInteger", ElementName = "PosInt")]
    public string PositiveInt;
    // This is ignored when serialized unless it is overridden.
    [SoapIgnore]
    public bool IgnoreThis;

    public GroupType GroupType;

    [SoapInclude(typeof(Car))]
    public Vehicle myCar(string licNumber)
    {
        Vehicle v;
        if(licNumber == "")
        {
            v = new Car();
            v.licenseNumber = "!!!!!!";
        }
        else
        {
            v = new Car();
            v.licenseNumber = licNumber;
        }
        return v;
    }

    public abstract class Vehicle
    {
        public string licenseNumber;
        public DateTime makeDate;
    }

    public class Car: Vehicle
```

```
{  
}  
  
public enum GroupType  
{  
    // These enums can be overridden.  
    small,  
    large  
}  
  
public class Run  
{  
    public static void Main()  
    {  
        Run test = new Run();  
        test.SerializeOriginal("SoapOriginal.xml");  
        test.SerializeOverride("SoapOverrides.xml");  
        test.DeserializeOriginal("SoapOriginal.xml");  
        test.DeserializeOverride("SoapOverrides.xml");  
  
    }  
    public void SerializeOriginal(string filename)  
    {  
        // Creates an instance of the XmlSerializer class.  
        XmlTypeMapping myMapping =  
            (new SoapReflectionImporter().ImportTypeMapping(  
                typeof(Group)));  
        XmlSerializer mySerializer =  
            new XmlSerializer(myMapping);  
  
        // Writing the file requires a TextWriter.  
        TextWriter writer = new StreamWriter(filename);  
  
        // Creates an instance of the class that will be serialized.  
        Group myGroup = new Group();  
  
        // Sets the object properties.  
        myGroup.GroupName = ".NET";  
  
        Byte [] hexByte = new Byte[2]{Convert.ToByte(100),  
            Convert.ToByte(50)};  
        myGroup.GroupNumber = hexByte;  
  
        DateTime myDate = new DateTime(2002,5,2);  
        myGroup.Today = myDate;  
  
        myGroup.PositiveInt= "10000";  
        myGroup.IgnoreThis=true;  
        myGroup.GroupType= GroupType.small;  
        Car thisCar =(Car) myGroup.myCar("1234566");  
  
        // Prints the license number just to prove the car was created.  
        Console.WriteLine("License#:" + thisCar.licenseNumber + "\n");  
  
        // Serializes the class and closes the TextWriter.  
    }  
}
```

```
        mySerializer.Serialize(writer, myGroup);
        writer.Close();
    }

    public void SerializeOverride(string filename)
    {
        // Creates an instance of the XmlSerializer class
        // that overrides the serialization.
        XmlSerializer overRideSerializer = CreateOverrideSerializer();

        // Writing the file requires a TextWriter.
        TextWriter writer = new StreamWriter(filename);

        // Creates an instance of the class that will be serialized.
        Group myGroup = new Group();

        // Sets the object properties.
        myGroup.GroupName = ".NET";

        Byte [] hexByte = new Byte[2]{Convert.ToByte(100),
        Convert.ToByte(50)};
        myGroup.GroupNumber = hexByte;

        DateTime myDate = new DateTime(2002,5,2);
        myGroup.Today = myDate;

        myGroup.PositiveInt= "10000";
        myGroup.IgnoreThis=true;
        myGroup.GroupType= GroupType.small;
        Car thisCar =(Car) myGroup.myCar("1234566");

        // Serializes the class and closes the TextWriter.
        overRideSerializer.Serialize(writer, myGroup);
        writer.Close();
    }

    public void DeserializeOriginal(string filename)
    {
        // Creates an instance of the XmlSerializer class.
        XmlTypeMapping myMapping =
        (new SoapReflectionImporter().ImportTypeMapping(
        typeof(Group)));
        XmlSerializer mySerializer =
        new XmlSerializer(myMapping);

        TextReader reader = new StreamReader(filename);

        // Deserializes and casts the object.
        Group myGroup;
        myGroup = (Group) mySerializer.Deserialize(reader);

        Console.WriteLine(myGroup.GroupName);
        Console.WriteLine(myGroup.GroupNumber[0]);
        Console.WriteLine(myGroup.GroupNumber[1]);
        Console.WriteLine(myGroup.Today);
    }
}
```

```
        Console.WriteLine(myGroup.PositiveInt);
        Console.WriteLine(myGroup.IgnoreThis);
        Console.WriteLine();
    }

    public void DeserializeOverride(string filename)
    {
        // Creates an instance of the XmlSerializer class.
        XmlSerializer overRideSerializer = CreateOverrideSerializer();
        // Reading the file requires a TextReader.
        TextReader reader = new StreamReader(filename);

        // Deserializes and casts the object.
        Group myGroup;
        myGroup = (Group) overRideSerializer.Deserialize(reader);

        Console.WriteLine(myGroup.GroupName);
        Console.WriteLine(myGroup.GroupNumber[0]);
        Console.WriteLine(myGroup.GroupNumber[1]);
        Console.WriteLine(myGroup.Today);
        Console.WriteLine(myGroup.PositiveInt);
        Console.WriteLine(myGroup.IgnoreThis);
    }

    private XmlSerializer CreateOverrideSerializer()
    {
        SoapAttributeOverrides mySoapAttributeOverrides =
            new SoapAttributeOverrides();
        SoapAttributes soapAtts = new SoapAttributes();

        SoapElementAttribute mySoapElement = new SoapElementAttribute();
        mySoapElement.ElementName = "xxxx";
        soapAtts.SoapElement = mySoapElement;
        mySoapAttributeOverrides.Add(typeof(Group), "PositiveInt",
            soapAtts);

        // Overrides the IgnoreThis property.
        SoapIgnoreAttribute myIgnore = new SoapIgnoreAttribute();
        soapAtts = new SoapAttributes();
        soapAtts.SoapIgnore = false;
        mySoapAttributeOverrides.Add(typeof(Group), "IgnoreThis",
            soapAtts);

        // Overrides the GroupType enumeration.
        soapAtts = new SoapAttributes();
        SoapEnumAttribute xSoapEnum = new SoapEnumAttribute();
        xSoapEnum.Name = "Over1000";
        soapAtts.SoapEnum = xSoapEnum;

        // Adds the SoapAttributes to the
        // mySoapAttributeOverrides.
        mySoapAttributeOverrides.Add(typeof(GroupType), "large",
            soapAtts);

        // Creates a second enumeration and adds it.
    }
}
```

```
        soapAtts = new SoapAttributes();
        xSoapEnum = new SoapEnumAttribute();
        xSoapEnum.Name = "ZeroTo1000";
        soapAtts.SoapEnum = xSoapEnum;
        mySoapAttributeOverrides.Add(typeof(GroupType), "small",
        soapAtts);

        // Overrides the Group type.
        soapAtts = new SoapAttributes();
        SoapTypeAttribute soapType = new SoapTypeAttribute();
        soapType.TypeName = "Team";
        soapAtts.SoapType = soapType;
        mySoapAttributeOverrides.Add(typeof(Group), soapAtts);

        // Creates an XmlTypeMapping that is used to create an instance
        // of the XmlSerializer class. Then returns the XmlSerializer.
        XmlTypeMapping myMapping = (new SoapReflectionImporter(
        mySoapAttributeOverrides)).ImportTypeMapping(typeof(Group));

        XmlSerializer ser = new XmlSerializer(myMapping);
        return ser;
    }
}
```

See also

- [XML and SOAP Serialization](#)
- [Attributes That Control Encoded SOAP Serialization](#)
- [XML Serialization with XML Web Services](#)
- [How to: Serialize an Object](#)
- [How to: Deserialize an Object](#)
- [How to: Serialize an Object as a SOAP-Encoded XML Stream](#)

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

How to: chunk serialized data

Article • 04/05/2023

⚠ Warning

Binary serialization can be dangerous. For more information, see [BinaryFormatter security guide](#).

Two issues that occur when sending large data sets in Web service messages are:

1. A large working set (memory) due to buffering by the serialization engine.
2. Inordinate bandwidth consumption due to 33 percent inflation after Base64 encoding.

To solve these problems, implement the [IXmlSerializable](#) interface to control the serialization and deserialization. Specifically, implement the [WriteXml](#) and [ReadXml](#) methods to chunk the data.

To implement server-side chunking

1. On the server machine, the Web method must turn off ASP.NET buffering and return a type that implements [IXmlSerializable](#).
2. The type that implements [IXmlSerializable](#) chunks the data in the [WriteXml](#) method.

To implement client-side processing

1. Alter the Web method on the client proxy to return the type that implements [IXmlSerializable](#). You can use a [SchemaImporterExtension](#) to do this automatically, but this isn't shown here.
2. Implement the [ReadXml](#) method to read the chunked data stream and write the bytes to disk. This implementation also raises progress events that can be used by a graphic control, such as a progress bar.

Example

The following code example shows the Web method on the client that turns off ASP.NET buffering. It also shows the client-side implementation of the [IXmlSerializable](#) interface that chunks the data in the [WriteXml](#) method.

C#

```
[WebMethod]
[SoapDocumentMethod(ParameterStyle = SoapParameterStyle.Bare)]
public SongStream DownloadSong(DownloadAuthorization Authorization, string
filePath)
{
    // Turn off response buffering.
    System.Web.HttpContext.Current.Response.Buffer = false;
    // Return a song.
    SongStream song = new SongStream(filePath);
    return song;
}
```

C#

```
[XmlSchemaProvider("MySchema")]
public class SongStream : IXmlSerializable
{
    private const string ns = "http://demos.Contoso.com/webservices";
    private string filePath;

    public SongStream() { }

    public SongStream(string filePath)
    {
        this.filePath = filePath;
    }

    // This is the method named by the XmlSchemaProviderAttribute applied to
    // the type.
    public static XmlQualifiedName MySchema(XmlSchemaSet xs)
    {
        // This method is called by the framework to get the schema for this
        // type.
        // We return an existing schema from disk.

        XmlSerializer schemaSerializer = new
        XmlSerializer(typeof(XmlSchema));
        string xsdPath = null;
        // NOTE: replace the string with your own path.
        xsdPath =
System.Web.HttpContext.Current.Server.MapPath("SongStream.xsd");
        XmlSchema s = (XmlSchema)schemaSerializer.Deserialize(
            new XmlTextReader(xsdPath), null);
        xs.XmlResolver = new XmlUrlResolver();
        xs.Add(s);
    }
}
```

```

        return new XmlQualifiedName("songStream", ns);
    }

    void IXmlSerializable.WriteXml(System.Xml.XmlWriter writer)
    {
        // This is the chunking code.
        // ASP.NET buffering must be turned off for this to work.

        int bufferSize = 4096;
        char[] songBytes = new char[bufferSize];
        FileStream inFile = File.Open(this.filePath, FileMode.Open,
FileAccess.Read);

        long length = inFile.Length;

        // Write the file name.
        writer.WriteElementString("fileName", ns,
Path.GetFileNameWithoutExtension(this.filePath));

        // Write the size.
        writer.WriteElementString("size", ns, length.ToString());

        // Write the song bytes.
        writer.WriteStartElement("song", ns);

        StreamReader sr = new StreamReader(inFile, true);
        int readLen = sr.Read(songBytes, 0, bufferSize);

        while (readLen > 0)
        {
            writer.WriteStartElement("chunk", ns);
            writer.WriteChars(songBytes, 0, readLen);
            writer.WriteEndElement();

            writer.Flush();
            readLen = sr.Read(songBytes, 0, bufferSize);
        }

        writer.WriteEndElement();
        inFile.Close();
    }

    XmlSchema IXmlSerializable.GetSchema()
    {
        throw new NotImplementedException();
    }

    void IXmlSerializable.ReadXml(System.Xml.XmlReader reader)
    {
        throw new NotImplementedException();
    }
}

```

C#

```
public class SongFile : IXmlSerializable
{
    public static event ProgressMade OnProgress;

    public SongFile()
    { }

    private const string ns = "http://demos.teched2004.com/webservices";
    public static string MusicPath;
    private string filePath;
    private double size;

    void IXmlSerializable.ReadXml(System.Xml.XmlReader reader)
    {
        reader.ReadStartElement("DownloadSongResult", ns);
        ReadFileName(reader);
        ReadSongSize(reader);
        ReadAndSaveSong(reader);
        reader.ReadEndElement();
    }

    void ReadFileName(XmlReader reader)
    {
        string fileName = reader.ReadElementString("fileName", ns);
        this.filePath =
            Path.Combine(MusicPath, Path.ChangeExtension(fileName, ".mp3"));
    }

    void ReadSongSize(XmlReader reader)
    {
        this.size = Convert.ToDouble(reader.ReadElementString("size", ns));
    }

    void ReadAndSaveSong(XmlReader reader)
    {
        FileStream outFile = File.Open(
            this.filePath, FileMode.Create, FileAccess.Write);

        string songBase64;
        byte[] songBytes;
        reader.ReadStartElement("song", ns);
        double totalRead = 0;
        while (true)
        {
            if (reader.IsStartElement("chunk", ns))
            {
                songBase64 = reader.ReadElementString();
                totalRead += songBase64.Length;
                songBytes = Convert.FromBase64String(songBase64);
                outFile.Write(songBytes, 0, songBytes.Length);
                outFile.Flush();
            }
        }
    }
}
```

```
        if (OnProgress != null)
    {
        OnProgress(100 * (totalRead / size));
    }
}

else
{
    break;
}
}

outFile.Close();
reader.ReadEndElement();
}

public void Play()
{
    System.Diagnostics.Process.Start(this.filePath);
}

XmlSchema IXmlSerializable.GetSchema()
{
    throw new NotImplementedException();
}

public void WriteXml(XmlWriter writer)
{
    throw new NotImplementedException();
}
}
```

Compiling the code

- The code uses the following namespaces: [System](#), [System.Runtime.Serialization](#), [System.Web.Services](#), [System.Web.Services.Protocols](#), [System.Xml](#), [System.Xml.Serialization](#), and [System.Xml.Schema](#).

See also

- [Custom Serialization](#)

 Collaborate with us on
GitHub

.NET

.NET feedback

The .NET documentation is open source. Provide feedback [here](#).

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

 [Open a documentation issue](#)

 [Provide product feedback](#)

<system.xml.serialization> Element

Article • 09/15/2021

The top-level element for controlling XML serialization. For more information about configuration files, see [Configuration File Schema](#).

```
<configuration>
<system.xml.serialization>
```

Syntax

XML

```
<system.xml.serialization>
</system.xml.serialization>
```

Attributes and Elements

The following sections describe attributes, child elements, and parent elements.

Attributes

None.

Child Elements

| Element | Description |
|--|---|
| <code><dateTimeSerialization> Element</code> | Determines the serialization mode of <code>DateTime</code> objects. |
| <code><schemalimporterExtensions> Element</code> | Contains types that are used by the <code>XmlSchemalimporter</code> for mapping of XSD types to .NET types. |

Parent Elements

| Element | Description |
|--|---|
| <code><configuration> Element</code> | The root element in every configuration file that is used by the common language runtime and .NET Framework applications. |

Example

The following code example illustrates how to specify the serialization mode of a `DateTime` object, and the addition of types used by the `XmlSchemaImporter` when mapping XSD types to .NET types.

XML

```
<system.xml.serialization>
  <xmleserializer checkDeserializeAdvances="false" />
  <dateTimeSerialization mode = "Local" />
  <schemaImporterExtensions>
    <add
      name = "MobileCapabilities"
      type = "System.Web.Mobile.MobileCapabilities,
      System.Web.Mobile, Version=2.0.0.0, Culture=neutral,
      PublicKeyToken=b03f5f6f11d40a3a" />
  </schemaImporterExtensions>
</system.xml.serialization>
```

See also

- [XmlSchemaImporter](#)
- [DateTimeSerializationSection.DateTimeSerializationMode](#)
- [Configuration File Schema](#)
- [<dateTimeSerialization> Element](#)
- [<schemaimporterExtensions> Element](#)
- [<add> Element for <schemaimporterExtensions>](#)

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

<dateTimeSerialization> Element

Article • 09/15/2021

Determines the serialization mode of [DateTime](#) objects.

```
<configuration>
  <dateTimeSerialization>
```

Syntax

XML

```
<dateTimeSerialization
  mode = "Roundtrip|Local"
/>
```

Attributes and Elements

The following sections describe attributes, child elements, and parent elements.

Attributes

| Attributes | Description |
|------------|---|
| mode | Optional. Specifies the serialization mode. Set to one of the DateTimeSerializationSection.DateTimeSerializationMode values. The default is RoundTrip . |

Child Elements

None.

Parent Elements

| Element | Description |
|--------------------------|--|
| system.xml.serialization | The top-level element for controlling XML serialization. |

Remarks

When this property is set to **Local**, [DateTime](#) objects are always formatted as the local time. That is, local time zone information is always included with the serialized data.

When this property is set to **Roundtrip**, [DateTime](#) objects are examined to determine whether they are in the local, UTC, or an unspecified time zone. The [DateTime](#) objects are then serialized in such a way that this information is preserved. This is the default behavior and is the recommended behavior for all new applications that do not communicate with older versions of the framework.

See also

- [DateTime](#)
- [XmlSchemaImporter](#)
- [DateTimeSerializationSection.DateTimeSerializationMode](#)
- [Configuration File Schema](#)
- [<schemainporterExtensions> Element](#)
- [<add> Element for <schemainporterExtensions>](#)
- [<system.xml.serialization> Element](#)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback [here](#).

 [Open a documentation issue](#)

 [Provide product feedback](#)

<schemaImporterExtensions> element

Article • 09/15/2021

Contains types that are used by the [XmlSchemaImporter](#) for mapping of XSD types to .NET types. For more information about configuration files, see [Configuration File Schema](#).

Syntax

XML

```
<schemaImporterExtensions>
    <!-- Add types -->
</schemaImporterExtensions>
```

Child Elements

| Element | Description |
|--|---|
| <add> Element for <schemaImporterExtensions> | Adds types that are used by the XmlSchemaImporter to create mappings. |

Parent Elements

| Element | Description |
|--|--|
| <system.xml.serialization> Element | The top-level element for controlling XML serialization. |

Example

The following code example illustrates how to add types that are used by the [XmlSchemaImporter](#) when mapping XSD types to .NET types.

XML

```
<system.xml.serialization>
    <schemaImporterExtensions>
        <add name = "MobileCapabilities" type =
            "System.Web.Mobile.MobileCapabilities,
            System.Web.Mobile, Version = 2.0.0.0, Culture = neutral,
```

```
    PublicKeyToken = b03f5f6f11d40a3a" />
  </schemaImporterExtensions>
</system.xml.serialization>
```

See also

- [XmlSchemaImporter](#)
- [DateTimeSerializationSection.DateTimeSerializationMode](#)
- [Configuration File Schema](#)
- [<dateTimeSerialization> Element](#)
- [<add> Element for <schemainporterExtensions>](#)
- [<system.xml.serialization> Element](#)

 [Collaborate with us on GitHub](#)

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

<add> Element for <schemalimporterExtensions>

Article • 09/15/2021

Adds types used by the [XmlSchemalimporter](#) for mapping XSD types to .NET types. For more information about configuration files, see [Configuration File Schema](#).

```
<configuration>
<system.xml.serialization>
<schemalimporterExtensions>
<add>
```

Syntax

XML

```
<add name = "typeName" type="fully qualified type [,Version=version number]
[,Culture=culture] [,PublicKeyToken= token]" />
```

Attributes and Elements

The following sections describe attributes, child elements, and parent elements.

Attributes

| Attribute | Description |
|-----------|--|
| name | A simple name that is used to find the instance. |
| type | Required. Specifies the schema extension class to add. The type attribute value must be on one line, and include the fully qualified type name. When the assembly is placed in the Global Assembly Cache (GAC), it must also include the version, culture, and public key token of the signed assembly. |

Child Elements

None.

Parent Elements

| Element | Description |
|-----------------------------|--|
| <schemalimporterExtensions> | Contains the types that are used by the XmlSchemalimporter . |

Example

The following code example adds an extension type that the XmlSchemalimporter can use when mapping types.

XML

```
<configuration>
  <system.xml.serialization>
    <schemalimporterExtensions>
      <add name="contoso" type="System.Web.Mobile.MobileCapabilities,
        System.Web.Mobile, Version=2.0.0.0, Culture=neutral,
        PublicKeyToken=b03f5f7f11d50a3a" />
    </schemalimporterExtensions>
  </system.xml.serialization>
</configuration>
```

See also

- [XmlSchemalimporter](#)
- [<system.xml.serialization> Element](#)
- [<schemalimporterExtensions> Element](#)

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback [here](#).

 [Open a documentation issue](#)

 [Provide product feedback](#)

<xmlSerializer> Element

Article • 09/15/2021

Specifies whether an additional check of progress of the [XmlSerializer](#) is done.

```
<configuration>
<system.xml.serialization>
```

Syntax

XML

```
<xmlSerializer checkDeserializerAdvance = "true|false" />
```

Attributes and Elements

The following sections describe attributes, child elements, and parent elements.

Attributes

| Attribute | Description |
|----------------------------------|--|
| checkDeserializeAdvances | Specifies whether the progress of the XmlSerializer is checked. Set the attribute to "true" or "false". The default is "true". |
| useLegacySerializationGeneration | Specifies whether the XmlSerializer uses legacy serialization generation which generates assemblies by writing C# code to a file and then compiling it to an assembly. The default is false. |

Child Elements

None.

Parent Elements

| Element | Description |
|----------------------------|---|
| <system.xml.serialization> | Contains configuration settings for the XmlSerializer and |

| Element | Description |
|---------|---|
| Element | <code>XmlSchemaImporter</code> classes. |

Remarks

By default, the `XmlSerializer` provides an additional layer of security against potential denial of service attacks when deserializing untrusted data. It does so by attempting to detect infinite loops during deserialization. If such a condition is detected, an exception is thrown with the following message: "Internal error: deserialization failed to advance over underlying stream."

Receiving this message does not necessarily indicate that a denial of service attack is in progress. In some rare circumstances, the infinite loop detection mechanism produces a false positive and the exception is thrown for a legitimate incoming message. If you find that in your particular application legitimate messages are being rejected by this extra layer of protection, set `checkDeserializeAdvances` attribute to "false".

Example

The following code example sets the `checkDeserializeAdvances` attribute to "false".

XML

```
<configuration>
  <system.xml.serialization>
    <xmlSerializer checkDeserializeAdvances="false" />
  </system.xml.serialization>
</configuration>
```

See also

- [XmlSerializer](#)
- [<system.xml.serialization> Element](#)
- [XML and SOAP Serialization](#)

 Collaborate with us on
GitHub

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

 [Open a documentation issue](#)

 [Provide product feedback](#)

XML Serializer Generator Tool (Sgen.exe)

Article • 08/12/2022

The XML Serializer Generator creates an XML serialization assembly for types in a specified assembly. The serialization assembly improves the startup performance of a [XmlSerializer](#) when it serializes or deserializes objects of the specified types.

Syntax

Run the tool from the command line.

Console

```
sgen [options]
```

Tip

For .NET Framework tools to function properly, you must either use [Visual Studio Developer Command Prompt](#) or [Visual Studio Developer PowerShell](#) or set the `Path`, `Include`, and `Lib` environment variables correctly. To set these environment variables, run `SDKVars.bat`, which is located in the `<SDK>\<version>\Bin` directory.

Parameters

| Option | Description |
|----------------------------------|---|
| <code>/a[sembly]:filename</code> | Generates serialization code for all the types contained in the assembly or executable specified by <i>filename</i> . Only one file name can be provided. If this argument is repeated, the last file name is used. |
| <code>/c[ompiler]:options</code> | Specifies the options to pass to the C# compiler. All csc.exe options are supported as they are passed to the compiler. This can be used to specify that the assembly should be signed and to specify the key file. |
| <code>/d[ebug]</code> | Generates an image that can be used with a debugger. |
| <code>/f[orce]</code> | Forces the overwriting of an existing assembly of the same name. The default is false . |

| Option | Description |
|----------------------------|---|
| /help or /? | Displays command syntax and options for the tool. |
| /k[eepl] | Suppresses the deletion of the generated source files and other temporary files after they have been compiled into the serialization assembly. This can be used to determine whether the tool is generating serialization code for a particular type. |
| /n[ologo] | Suppresses the display of the Microsoft startup banner. |
| /o[ut]:path | Specifies the directory in which to save the generated assembly. Note: The name of the generated assembly is composed of the name of the input assembly plus "xmlSerializers.dll". |
| /p[roxytypes] | Generates serialization code only for the XML Web service proxy types. |
| /r[eference]:assemblyfiles | Specifies the assemblies that are referenced by the types requiring XML serialization. Accepts multiple assembly files separated by commas. |
| /s[ilent] | Suppresses the display of success messages. |
| /t[type]:type | Generates serialization code only for the specified type. |
| /v[erbose] | Displays verbose output for debugging. Lists types from the target assembly that cannot be serialized with the XmlSerializer . |
| /? | Displays command syntax and options for the tool. |

Remarks

When the XML Serializer Generator is not used, a [XmlSerializer](#) generates serialization code and a serialization assembly for each type every time an application is run. To improve the performance of XML serialization startup, use the Sgen.exe tool to generate those assemblies in advance. These assemblies can then be deployed with the application.

The XML Serializer Generator can also improve the performance of clients that use XML Web service proxies to communicate with servers because the serialization process will not incur a performance hit when the type is loaded the first time.

These generated assemblies cannot be used on the server side of a Web service. This tool is only for Web service clients and manual serialization scenarios.

If the assembly containing the type to serialize is named MyType.dll, then the associated serialization assembly will be named MyType.XmlSerializers.dll.

Note

The `sgen` tool is not compatible with `init-only` setters. The tool will fail if the target assembly contains any public properties that use this feature.

Examples

The following command creates an assembly named `Data.XmlSerializers.dll` for serializing all the types contained in the assembly named `Data.dll`.

Console

```
sgen Data.dll
```

The `Data.XmlSerializers.dll` assembly can be referenced from code that needs to serialize and deserialize the types in `Data.dll`.

See also

- [Tools](#)
- [Developer command-line shells](#)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback [here](#).

 [Open a documentation issue](#)

 [Provide product feedback](#)

XML Schema Definition Tool (Xsd.exe)

Article • 07/29/2022

The XML Schema Definition (Xsd.exe) tool generates XML schema or common language runtime classes from XDR, XML, and XSD files, or from classes in a runtime assembly.

The XML Schema Definition tool (Xsd.exe) usually can be found in the following path:
C:\Program Files (x86)\Microsoft SDKs\Windows\{version}\bin\NETFX {version} Tools

Syntax

Run the tool from the command line.

Console

```
xsd file.xdr [-outputdir:directory][/parameters:file.xml]
xsd file.xml [-outputdir:directory] [/parameters:file.xml]
xsd file.xsd{/classes | /dataset} [/element:element]
    [/enableLinqDataSet] [/language:language]
    [/namespace:namespace] [-outputdir:directory]
[URI:uri]
    [/parameters:file.xml]
xsd {file.dll | file.exe} [-outputdir:directory] [/type:typename [...]]
[/parameters:file.xml]
```

Tip

For .NET Framework tools to function properly, you must set your `Path`, `Include`, and `Lib` environment variables correctly. Set these environment variables by running `SDKVars.bat`, which is located in the `<SDK>\<version>\Bin` directory. `SDKVars.bat` must be executed in every command shell.

Argument

| Argument | Description |
|-----------------------------|--|
| <code>file.extension</code> | Specifies the input file to convert. You must specify the extension as one of the following: <code>.xdr</code> , <code>.xml</code> , <code>.xsd</code> , <code>.dll</code> , or <code>.exe</code> . If you specify an XDR schema file (<code>.xdr</code> extension), <code>Xsd.exe</code> converts the XDR schema to an XSD schema. The output file has the same name as the XDR schema, but with the <code>.xsd</code> extension. |

| Argument | Description |
|----------|---|
| | If you specify an XML file (.xml extension), Xsd.exe infers a schema from the data in the file and produces an XSD schema. The output file has the same name as the XML file, but with the .xsd extension. |
| | If you specify an XML schema file (.xsd extension), Xsd.exe generates source code for runtime objects that correspond to the XML schema. |
| | If you specify a runtime assembly file (.exe or .dll extension), Xsd.exe generates schemas for one or more types in that assembly. You can use the <code>/type</code> option to specify the types for which to generate schemas. The output schemas are named schema0.xsd, schema1.xsd, and so on. Xsd.exe produces multiple schemas only if the given types specify a namespace using the <code>XMLRoot</code> custom attribute. |

General Options

| Option | Description |
|-------------------------------------|--|
| <code>/h[elp]</code> | Displays command syntax and options for the tool. |
| <code>/o[utputdir]:directory</code> | Specifies the directory for output files. This argument can appear only once. The default is the current directory. |
| <code>/?</code> | Displays command syntax and options for the tool. |
| <code>/p[arameters]:file.xml</code> | Read options for various operation modes from the specified .xml file. The short form is <code>/p:</code> . For more information, see the Remarks section. |

XSD File Options

You must specify only one of the following options for .xsd files.

| Option | Description |
|-------------------------|---|
| <code>/c[lasses]</code> | Generates classes that correspond to the specified schema. To read XML data into the object, use the XmlSerializer.Deserialize method. |
| <code>/d[ataset]</code> | Generates a class derived from DataSet that corresponds to the specified schema. To read XML data into the derived class, use the DataSet.ReadXml method. |

You can also specify any of the following options for .xsd files.

| Option | Description |
|------------------------|---|
| /e[element]:element | Specifies the element in the schema to generate code for. By default all elements are typed. You can specify this argument more than once. |
| /enableDataBinding | Implements the INotifyPropertyChanged interface on all generated types to enable data binding. The short form is <code>/edb</code> . |
| /enableLinqDataSet | (Short form: <code>/eld</code> .) Specifies that the generated DataSet can be queried against using LINQ to DataSet. This option is used when the <code>/dataset</code> option is also specified. For more information, see LINQ to DataSet Overview and Querying Typed DataSets . For general information about using LINQ, see Language-Integrated Query (LINQ) - C# or Language-Integrated Query (LINQ) - Visual Basic . |
| /f[ields] | Generates fields only. By default, properties with backing fields are generated. |
| /l[anguage]:language | Specifies the programming language to use. Choose from <code>cs</code> (C#, which is the default), <code>vb</code> (Visual Basic), <code>js</code> (JScript), or <code>vjs</code> (Visual J#). You can also specify a fully qualified name for a class implementing System.CodeDom.Compiler.CodeDomProvider |
| /n[amespace]:namespace | Specifies the runtime namespace for the generated types. The default namespace is <code>Schemas</code> . |
| /nologo | Suppresses the banner. |
| /order | Generates explicit order identifiers on all particle members. |
| /o[ut]:directoryName | Specifies the output directory to place the files in. The default is the current directory. |
| /u[ri]:uri | Specifies the URI for the elements in the schema to generate code for. This URI, if present, applies to all elements specified with the <code>/element</code> option. |

DLL and EXE File Options

| Option | Description |
|-------------------|---|
| /t[type]:typename | Specifies the name of the type to create a schema for. You can specify multiple type arguments. If <code>typename</code> does not specify a namespace, Xsd.exe matches all types in the assembly with the specified type. If <code>typename</code> specifies a namespace, only that type is matched. If <code>typename</code> ends with an asterisk character (*), the tool matches all types that start with the string preceding the *. If you omit the <code>/type</code> option, Xsd.exe generates schemas for all types in the assembly. |

Remarks

The following table shows the operations that Xsd.exe performs.

| Operation | Description |
|----------------|---|
| XDR to XSD | Generates an XML schema from an XML-Data-Reduced schema file. XDR is an early XML-based schema format. |
| XML to XSD | Generates an XML schema from an XML file. |
| XSD to DataSet | Generates common language runtime DataSet classes from an XSD schema file. The generated classes provide a rich object model for regular XML data. |
| XSD to Classes | Generates runtime classes from an XSD schema file. The generated classes can be used in conjunction with System.Xml.Serialization.XmlSerializer to read and write XML code that follows the schema. |
| Classes to XSD | Generates an XML schema from a type or types in a runtime assembly file. The generated schema defines the XML format used by the XmlSerializer . |

Xsd.exe only allows you to manipulate XML schemas that follow the XML Schema Definition (XSD) language proposed by the World Wide Web Consortium (W3C). For more information on the XML Schema Definition proposal or the XML standard, see <https://w3.org>.

Setting Options with an XML File

By using the `/parameters` switch, you can specify a single XML file that sets various options. The options you can set depend on how you are using the XSD.exe tool. Choices include generating schemas, generating code files, or generating code files that include `DataSet` features. For example, you can set the `<assembly>` element to the name of an executable (.exe) or type library (.dll) file when generating a schema, but not when generating a code file. The following XML shows how to use the `<generateSchemas>` element with a specified executable:

XML

```
<!-- This is in a file named GenerateSchemas.xml. -->
<xsd xmlns='http://microsoft.com/dotnet/tools/xsd/'>
<generateSchemas>
    <assembly>ConsoleApplication1.exe</assembly>
</generateSchemas>
</xsd>
```

If the preceding XML is contained in a file named GenerateSchemas.xml, then use the `/parameters` switch by typing the following at a command prompt and pressing `Enter`:

Console

```
xsd /p:GenerateSchemas.xml
```

On the other hand, if you are generating a schema for a single type found in the assembly, you can use the following XML:

XML

```
<!-- This is in a file named GenerateSchemaFromType.xml. -->
<xsd xmlns='http://microsoft.com/dotnet/tools/xsd/'>
<generateSchemas>
  <type>IDItems</type>
</generateSchemas>
</xsd>
```

But to use preceding code, you must also supply the name of the assembly at the command prompt. Enter the following at a command prompt (presuming the XML file is named GenerateSchemaFromType.xml):

Console

```
xsd /p:GenerateSchemaFromType.xml ConsoleApplication1.exe
```

You must specify only one of the following options for the `<generateSchemas>` element.

| Element | Description |
|-------------------------------|---|
| <code><assembly></code> | Specifies an assembly to generate the schema from. |
| <code><type></code> | Specifies a type found in an assembly to generate a schema for. |
| <code><xml></code> | Specifies an XML file to generate a schema for. |
| <code><xdr></code> | Specifies an XDR file to generate a schema for. |

To generate a code file, use the `<generateClasses>` element. The following example generates a code file. Note that two attributes are also shown that allow you to set the programming language and namespace of the generated file.

XML

```

<xsd xmlns='http://microsoft.com/dotnet/tools/xsd/'>
<generateClasses language='VB'
namespace='Microsoft.Serialization.Examples' />
</xsd>
<!-- You must supply an .xsd file when typing in the command line.-->
<!-- For example: xsd /p:genClasses mySchema.xsd -->

```

Options you can set for the `<generateClasses>` element include the following.

| Element | Description |
|--|---|
| <code><element></code> | Specifies an element in the .xsd file to generate code for. |
| <code><schemalimporterExtensions></code> | Specifies a type derived from the SchemalimporterExtension class. |
| <code><schema></code> | Specifies a XML Schema file to generate code for. Multiple XML Schema files can be specified using multiple <code><schema></code> elements. |

The following table shows the attributes that can also be used with the `<generateClasses>` element.

| Attribute | Description |
|------------------------|--|
| <code>language</code> | Specifies the programming language to use. Choose from <code>cs</code> (C#, the default), <code>VB</code> (Visual Basic), <code>JS</code> (JScript), or <code>VJS</code> (Visual J#). You can also specify a fully qualified name for a class that implements CodeDomProvider . |
| <code>namespace</code> | Specifies the namespace for the generated code. The namespace must conform to CLR standards (for example, no spaces or backslash characters). |
| <code>options</code> | One of the following values: <code>none</code> , <code>properties</code> (generates properties instead of public fields), <code>order</code> , or <code>enableDataBinding</code> (see the <code>/order</code> and <code>/enableDataBinding</code> switches in the preceding XSD File Options section). |

You can also control how `DataSet` code is generated by using the `<generateDataSet>` element. The following XML specifies that the generated code uses `DataSet` structures (such as the `DataTable` class) to create Visual Basic code for a specified element. The generated DataSet structures will support LINQ queries.

XML

```

<xsd xmlns='http://microsoft.com/dotnet/tools/xsd/'>
  <generateDataSet language='VB'
  namespace='Microsoft.Serialization.Examples' enableLinqDataSet='true'>
    </generateDataSet>
  </xsd>

```

Options you can set for the `<generateDataSet>` element include the following.

| Element | Description |
|-----------------------------|--|
| <code><schema></code> | Specifies an XML Schema file to generate code for. Multiple XML Schema files can be specified using multiple <code><schema></code> elements. |

The following table shows the attributes that can be used with the `<generateDataSet>` element.

| Attribute | Description |
|--------------------------------|---|
| <code>enableLinqDataSet</code> | Specifies that the generated DataSet can be queried against using LINQ to DataSet. The default value is false. |
| <code>language</code> | Specifies the programming language to use. Choose from <code>cs</code> (C#, the default), <code>VB</code> (Visual Basic), <code>JS</code> (JScript), or <code>VJS</code> (Visual J#). You can also specify a fully qualified name for a class that implements CodeDomProvider . |
| <code>namespace</code> | Specifies the namespace for the generated code. The namespace must conform to CLR standards (for example, no spaces or backslash characters). |

There are attributes that you can set on the top level `<xsd>` element. These options can be used with any of the child elements (`<generateSchemas>`, `<generateClasses>` or `<generateDataSet>`). The following XML code generates code for an element named "IDItems" in the output directory named "MyOutputDirectory".

| XML |
|---|
| <pre><xsd xmlns='http://microsoft.com/dotnet/tools/xsd/' output='MyOutputDirectory'> <generateClasses> <element>IDItems</element> </generateClasses> </xsd></pre> |

The following table shows the attributes that can also be used with the `<xsd>` element.

| Attribute | Description |
|---------------------|--|
| <code>output</code> | The name of a directory where the generated schema or code file will be placed. |
| <code>nologo</code> | Suppresses the banner. Set to <code>true</code> or <code>false</code> . |
| <code>help</code> | Displays command syntax and options for the tool. Set to <code>true</code> or <code>false</code> . |

Examples

The following command generates an XML schema from `myFile.xdr` and saves it to the current directory.

```
Console
```

```
xsd myFile.xdr
```

The following command generates an XML schema from `myFile.xml` and saves it to the specified directory.

```
Console
```

```
xsd myFile.xml /outputdir:myOutputDir
```

The following command generates a data set that corresponds to the specified schema in the C# language and saves it as `XSDSchemaFile.cs` in the current directory.

```
Console
```

```
xsd /dataset /language:CS XSDSchemaFile.xsd
```

The following command generates XML schemas for all types in the assembly `myAssembly.dll` and saves them as `schema0.xsd` in the current directory.

```
Console
```

```
xsd myAssembly.dll
```

See also

- [DataSet](#)
- [System.Xml.Serialization.XmlSerializer](#)
- [Tools](#)
- [Developer command-line shells](#)
- [LINQ to DataSet Overview](#)
- [Querying Typed DataSets](#)
- [LINQ \(Language-Integrated Query\) \(C#\)](#)
- [LINQ \(Language-Integrated Query\) \(Visual Basic\)](#)

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

Deserialization risks in use of BinaryFormatter and related types

Article • 04/05/2023

This article applies to the following types:

- [BinaryFormatter](#)
- [SoapFormatter](#)
- [NetDataContractSerializer](#)
- [LosFormatter](#)
- [ObjectStateFormatter](#)

This article applies to the following .NET implementations:

- .NET Framework all versions
- .NET Core 2.1 - 3.1
- .NET 5 and later

⚠ Warning

The **BinaryFormatter** type is dangerous and is *not* recommended for data processing. Applications should stop using `BinaryFormatter` as soon as possible, even if they believe the data they're processing to be trustworthy. `BinaryFormatter` is insecure and can't be made secure.

Deserialization vulnerabilities

Deserialization vulnerabilities are a threat category where request payloads are processed insecurely. An attacker who successfully leverages these vulnerabilities against an app can cause denial of service (DoS), information disclosure, or remote code execution inside the target app. This risk category consistently makes the [OWASP Top 10](#). Targets include apps written in [a variety of languages](#), including C/C++, Java, and C#.

In .NET, the biggest risk target is apps that use the `BinaryFormatter` type to deserialize data. `BinaryFormatter` is widely used throughout the .NET ecosystem because of its power and its ease of use. However, this same power gives attackers the ability to influence control flow within the target app. Successful attacks can result in the attacker being able to run code within the context of the target process.

As a simpler analogy, assume that calling `BinaryFormatter.Deserialize` over a payload is the equivalent of interpreting that payload as a standalone executable and launching it.

BinaryFormatter security vulnerabilities

⚠ Warning

The `BinaryFormatter.Deserialize` method is **never** safe when used with untrusted input. We strongly recommend that consumers instead consider using one of the alternatives outlined later in this article.

`BinaryFormatter` was implemented before deserialization vulnerabilities were a well-understood threat category. As a result, the code does not follow modern best practices. The `Deserialize` method can be used as a vector for attackers to perform DoS attacks against consuming apps. These attacks might render the app unresponsive or result in unexpected process termination. This category of attack cannot be mitigated with a `SerializationBinder` or any other `BinaryFormatter` configuration switch. .NET considers this behavior to be *by design* and won't issue a code update to modify the behavior.

`BinaryFormatter.Deserialize` may be vulnerable to other attack categories, such as information disclosure or remote code execution. Utilizing features such as a custom `SerializationBinder` may be insufficient to properly mitigate these risks. The possibility exists that a novel vulnerability will be discovered for which .NET cannot practically publish a security update. Consumers should assess their individual scenarios and consider their potential exposure to these risks.

We recommend that `BinaryFormatter` consumers perform individual risk assessments on their apps. It is the consumer's sole responsibility to determine whether to utilize `BinaryFormatter`. If you're considering using it, you should risk-assess the security, technical, reputation, legal, and regulatory consequences.

Preferred alternatives

.NET offers several in-box serializers that can handle untrusted data safely:

- `XmlSerializer` and `DataContractSerializer` to serialize object graphs into and from XML. Do not confuse `DataContractSerializer` with `NetDataContractSerializer`.
- `BinaryReader` and `BinaryWriter` for XML and JSON.
- The `System.Text.Json` APIs to serialize object graphs into JSON.

Dangerous alternatives

Avoid the following serializers:

- [SoapFormatter](#)
- [LosFormatter](#)
- [NetDataContractSerializer](#)
- [ObjectStateFormatter](#)

The preceding serializers all perform unrestricted polymorphic deserialization and are dangerous, just like `BinaryFormatter`.

The risks of assuming data to be trustworthy

Frequently, an app developer might believe that they are processing only trusted input. The safe input case is true in some rare circumstances. But it's much more common that a payload crosses a trust boundary without the developer realizing it.

Consider an **on-prem server** where employees use a desktop client from their workstations to interact with the service. This scenario might be seen naively as a "safe" setup where utilizing `BinaryFormatter` is acceptable. However, this scenario presents a vector for malware that gains access to a single employee's machine to be able to spread throughout the enterprise. That malware can leverage the enterprise's use of `BinaryFormatter` to move laterally from the employee's workstation to the backend server. It can then exfiltrate the company's sensitive data. Such data could include trade secrets or customer data.

Consider also an app that uses `BinaryFormatter` to persist save state. This might at first seem to be a safe scenario, as reading and writing data on your own hard drive represents a minor threat. However, sharing documents across email or the internet is common, and most end users wouldn't perceive opening these downloaded files as risky behavior.

This scenario can be leveraged to nefarious effect. If the app is a game, users who share save files unknowingly place themselves at risk. The developers themselves can also be targeted. The attacker might email the developers' tech support, attaching a malicious data file and asking the support staff to open it. This kind of attack could give the attacker a foothold in the enterprise.

Another scenario is where the data file is stored in cloud storage and automatically synced between the user's machines. An attacker who is able to gain access to the cloud storage account can poison the data file. This data file will be automatically synced to

the user's machines. The next time the user opens the data file, the attacker's payload runs. Thus the attacker can leverage a cloud storage account compromise to gain full code execution permissions.

Consider an app that moves from a desktop-install model to a cloud-first model. This scenario includes apps that move from a desktop app or rich client model into a web-based model. Any threat models drawn for the desktop app aren't necessarily applicable to the cloud-based service. The threat model for the desktop app might dismiss a given threat as "not interesting for the client to attack itself." But that same threat might become interesting when it considers a remote user (the client) attacking the cloud service itself.

Note

In general terms, the intent of serialization is to transmit an object into or out of an app. A threat modeling exercise almost always marks this kind of data transfer as crossing a trust boundary.

See also

- [Binary serialization](#)
- [YSoSerial.Net](#) ↗ for research into how adversaries attack apps that utilize `BinaryFormatter`.
- General background on deserialization vulnerabilities:
 - [OWASP: Deserialization of Untrusted Data](#) ↗
 - [CWE-502: Deserialization of Untrusted Data](#) ↗

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

BinaryFormatter event source

Article • 04/19/2023

Starting with .NET 5, [BinaryFormatter](#) includes a built-in [EventSource](#) that gives you visibility into when an object serialization or deserialization is occurring. Apps can use [EventListener](#)-derived types to listen for these notifications and log them.

This functionality is not a substitute for a [SerializationBinder](#) or an [ISerializationSurrogate](#) and can't be used to modify the data being serialized or deserialized. Rather, this eventing system is intended to provide insight into the types being serialized or deserialized. It can also be used to detect unintended calls into the [BinaryFormatter](#) infrastructure, such as calls originating from third-party library code.

Description of events

The `BinaryFormatter` event source has the well-known name `System.Runtime.Serialization.Formatters.Binary.BinaryFormatterEventSource`. Listeners can subscribe to six events.

SerializationStart event (id = 10)

Raised when `BinaryFormatter.Serialize` has been called and has started the serialization process. This event is paired with the `SerializationEnd` event. The `SerializationStart` event can be called recursively if an object calls `BinaryFormatter.Serialize` within its own serialization routine.

This event doesn't contain a payload.

SerializationEnd event (id = 11)

Raised when `BinaryFormatter.Serialize` has completed its work. Each occurrence of `SerializationEnd` denotes the completion of the last unpaired `SerializationStart` event.

This event doesn't contain a payload.

SerializingObject event (id = 12)

Raised when `BinaryFormatter.Serialize` is in the process of serializing a non-primitive type. The `BinaryFormatter` infrastructure special-cases certain types (such as `string` and `int`) and doesn't raise this event when these types are encountered. This event is raised for user-defined types and other types that `BinaryFormatter` doesn't natively understand.

This event may be raised zero or more times between `SerializationStart` and `SerializationEnd` events.

This event contains a payload with one argument:

- `typeName` (`string`): The assembly-qualified name (see [Type.AssemblyQualifiedName](#)) of the type being serialized.

DeserializationStart event (id = 20)

Raised when `BinaryFormatter.Deserialize` has been called and has started the deserialization process. This event is paired with the `DeserializationEnd` event. The `DeserializationStart` event can be called recursively if an object calls `BinaryFormatter.Deserialize` within its own deserialization routine.

This event doesn't contain a payload.

DeserializationEnd event (id = 21)

Raised when `BinaryFormatter.Deserialize` has completed its work. Each occurrence of `DeserializationEnd` denotes the completion of the last unpaired `DeserializationStart` event.

This event doesn't contain a payload.

DeserializingObject event (id = 22)

Raised when `BinaryFormatter.Deserialize` is in the process of deserializing a non-primitive type. The `BinaryFormatter` infrastructure special-cases certain types (such as `string` and `int`) and doesn't raise this event when these types are encountered. This event is raised for user-defined types and other types that `BinaryFormatter` doesn't natively understand.

This event may be raised zero or more times between `DeserializationStart` and `DeserializationEnd` events.

This event contains a payload with one argument.

- `typeName` (`string`): The assembly-qualified name (see [Type.AssemblyQualifiedName](#)) of the type being deserialized.

[Advanced] Subscribing to a subset of notifications

Listeners who wish to subscribe to only a subset of notifications can choose which keywords to enable.

- `Serialization` = `(EventKeywords)1`: Raises the `SerializationStart`, `SerializationEnd`, and `SerializingObject` events.
- `Deserialization` = `(EventKeywords)2`: Raises the `DeserializationStart`, `DeserializationEnd`, and `DeserializingObject` events.

If no keyword filters are provided during `EventListener` registration, all events are raised.

For more information, see [System.Diagnostics.Tracing.EventKeywords](#).

Sample code

The following code:

- creates an `EventListener`-derived type that writes to `System.Console`,
- subscribes that listener to `BinaryFormatter`-produced notifications,
- serializes and deserializes a simple object graph using `BinaryFormatter`, and
- analyzes the events that have been raised.

C#

```
using System;
using System.Diagnostics.Tracing;
using System.IO;
using System.Runtime.Serialization.Formatters.Binary;

namespace BinaryFormatterEventSample
{
    class Program
    {
        static EventListener? _globalListener = null;

        static void Main(string[] args)
        {
            // First, set up the event listener.
            // Note: We assign it to a static field so that it doesn't get
```

```

GCed.

        // We also provide a callback that subscribes this listener to
all
        // events produced by the well-known BinaryFormatter source.

        _globalListener = new ConsoleEventListener();
        _globalListener.EventSourceCreated += (sender, args) =>
    {
        if (args.EventSource?.Name ==
"System.Runtime.Serialization.Formatters.Binary.BinaryFormatterEventSource")
    {
        ((EventListener?)sender)?
            .EnableEvents(args.EventSource,
EventLevel.LogAlways);
    }
};

// Next, create the Person object and serialize it.

Person originalPerson = new Person()
{
    FirstName = "Logan",
    LastName = "Edwards",
    FavoriteBook = new Book()
    {
        Title = "A Tale of Two Cities",
        Author = "Charles Dickens",
        Price = 10.25m
    }
};

byte[] serializedPerson = SerializePerson(originalPerson);

// Finally, deserialize the Person object.

Person rehydratedPerson = DeserializePerson(serializedPerson);

Console.WriteLine
    ($"Rehydrated person {rehydratedPerson.FirstName}"
{rehydratedPerson.LastName}");
Console.Write
    ($"Favorite book: {rehydratedPerson.FavoriteBook?.Title} ");
Console.Write
    ($"by {rehydratedPerson.FavoriteBook?.Author}, ");
Console.WriteLine
    ($"list price {rehydratedPerson.FavoriteBook?.Price}");
}

private static byte[] SerializePerson(Person p)
{
    MemoryStream memStream = new MemoryStream();
    BinaryFormatter formatter = new BinaryFormatter();
#pragma warning disable SYSLIB0011 // BinaryFormatter.Serialize is obsolete
    formatter.Serialize(memStream, p);
}

```

```
#pragma warning restore SYSLIB0011

        return memStream.ToArray();
    }

    private static Person DeserializePerson(byte[] serializedData)
    {
        MemoryStream memStream = new MemoryStream(serializedData);
        BinaryFormatter formatter = new BinaryFormatter();

#pragma warning disable SYSLIB0011 // Danger: BinaryFormatter.Deserialize is
insecure for untrusted input
        return (Person)formatter.Deserialize(memStream);
#pragma warning restore SYSLIB0011
    }
}

[Serializable]
public class Person
{
    public string? FirstName;
    public string? LastName;
    public Book? FavoriteBook;
}

[Serializable]
public class Book
{
    public string? Title;
    public string? Author;
    public decimal? Price;
}

// A sample EventListener that writes data to System.Console.
public class ConsoleEventListener : EventListener
{
    protected override void OnEventWritten(EventWrittenEventArgs eventData)
    {
        base.OnEventWritten(eventData);

        Console.WriteLine($"Event {eventData.EventName} (id={eventData.EventId}) received.");
        if (eventData.PayloadNames != null)
        {
            for (int i = 0; i < eventData.PayloadNames.Count; i++)
            {
                Console.WriteLine($"{eventData.PayloadNames[i]} = {eventData.Payload?[i]}");
            }
        }
    }
}
```

The preceding code produces output similar to the following example:

Output

```
Event SerializationStart (id=10) received.  
Event SerializingObject (id=12) received.  
typeName = BinaryFormatterEventSample.Person, BinaryFormatterEventSample,  
Version=1.0.0.0, Culture=neutral, PublicKeyToken=null  
Event SerializingObject (id=12) received.  
typeName = BinaryFormatterEventSample.Book, BinaryFormatterEventSample,  
Version=1.0.0.0, Culture=neutral, PublicKeyToken=null  
Event SerializationStop (id=11) received.  
Event DeserializationStart (id=20) received.  
Event DeserializingObject (id=22) received.  
typeName = BinaryFormatterEventSample.Person, BinaryFormatterEventSample,  
Version=1.0.0.0, Culture=neutral, PublicKeyToken=null  
Event DeserializingObject (id=22) received.  
typeName = BinaryFormatterEventSample.Book, BinaryFormatterEventSample,  
Version=1.0.0.0, Culture=neutral, PublicKeyToken=null  
Event DeserializationStop (id=21) received.  
Rehydrated person Logan Edwards  
Favorite book: A Tale of Two Cities by Charles Dickens, list price 10.25
```

In this sample, the console-based `EventListener` logs that serialization starts, instances of `Person` and `Book` are serialized, and then serialization completes. Similarly, once deserialization has started, instances of `Person` and `Book` are deserialized, and then deserialization completes.

The app then prints the values contained in the deserialized `Person` to demonstrate that the object did in fact serialize and deserialize properly.

See also

For more information on using `EventListener` to receive `EventSource`-based notifications, see [the `EventListener` class](#).

 Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

more information, see [our contributor guide](#).

System.Runtime.Serialization.DataContractAttribute class

Article • 01/08/2024

This article provides supplementary remarks to the reference documentation for this API.

Apply the [DataContractAttribute](#) attribute to types (classes, structures, or enumerations) that are used in serialization and deserialization operations by the [DataContractSerializer](#). If you send or receive messages by using the Windows Communication Foundation (WCF) infrastructure, you should also apply the [DataContractAttribute](#) to any classes that hold and manipulate data sent in messages. For more information about data contracts, see [Using Data Contracts](#).

You must also apply the [DataMemberAttribute](#) to any field, property, or event that holds values you want to serialize. By applying the [DataContractAttribute](#), you explicitly enable the [DataContractSerializer](#) to serialize and deserialize the data.

✖ Caution

You can apply the [DataMemberAttribute](#) to private fields. Be aware that the data returned by the field (even if it is private) is serialized and deserialized, and thus can be viewed or intercepted by a malicious user or process.

For more information about data contracts, see the topics listed in [Using Data Contracts](#).

Data contracts

A *data contract* is an abstract description of a set of fields with a name and data type for each field. The data contract exists outside of any single implementation to allow services on different platforms to interoperate. As long as the data passed between the services conforms to the same contract, all the services can process the data. This processing is also known as a *loosely coupled system*. A data contract is also similar to an interface in that the contract specifies how data must be delivered so that it can be processed by an application. For example, the data contract may call for a data type named "Person" that has two text fields, named "FirstName" and "LastName". To create a data contract, apply the [DataContractAttribute](#) to the class and apply the [DataMemberAttribute](#) to any fields or properties that must be serialized. When serialized, the data conforms to the data contract that is implicitly built into the type.

Note

A data contract differs significantly from an actual interface in its inheritance behavior. Interfaces are inherited by any derived types. When you apply the **DataContractAttribute** to a base class, the derived types do not inherit the attribute or the behavior. However, if a derived type has a data contract, the data members of the base class are serialized. However, you must apply the **DataMemberAttribute** to new members in a derived class to make them serializable.

XML schema documents and the SvcUtil tool

If you are exchanging data with other services, you must describe the data contract. For the current version of the **DataContractSerializer**, an XML schema can be used to define data contracts. (Other forms of metadata/description could be used for the same purpose.) To create an XML schema from your application, use the [ServiceModel Metadata Utility Tool \(Svcutil.exe\)](#) with the **/dconly** command line option. When the input to the tool is an assembly, by default, the tool generates a set of XML schemas that define all the data contract types found in that assembly. Conversely, you can also use the Svcutil.exe tool to create Visual Basic or C# class definitions that conform to the requirements of XML schemas that use constructs that can be expressed by data contracts. In this case, the **/dconly** command line option is not required.

If the input to the Svcutil.exe tool is an XML schema, by default, the tool creates a set of classes. If you examine those classes, you find that the **DataContractAttribute** has been applied. You can use those classes to create a new application to process data that must be exchanged with other services.

You can also run the tool against an endpoint that returns a Web Services Description Language (WSDL) document to automatically generate the code and configuration to create an Windows Communication Foundation (WCF) client. The generated code includes types that are marked with the **DataContractAttribute**.

Reuse existing types

A data contract has two basic requirements: a stable name and a list of members. The stable name consists of the namespace uniform resource identifier (URI) and the local name of the contract. By default, when you apply the **DataContractAttribute** to a class, it uses the class name as the local name and the class's namespace (prefixed with "<http://schemas.datacontract.org/2004/07/>") as the namespace URI. You can override

the defaults by setting the [Name](#) and [Namespace](#) properties. You can also change the namespace by applying the [ContractNamespaceAttribute](#) to the namespace. Use this capability when you have an existing type that processes data exactly as you require but has a different namespace and class name from the data contract. By overriding the default values, you can reuse your existing type and have the serialized data conform to the data contract.

 **Note**

In any code, you can use the word `DataContract` instead of the longer `DataContractAttribute`.

Versioning

A data contract can also accommodate later versions of itself. That is, when a later version of the contract includes extra data, that data is stored and returned to a sender untouched. To do this, implement the [IExtensibleDataObject](#) interface.

For more information about versioning, see [Data Contract Versioning](#).

 **Collaborate with us on GitHub**

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

 .NET

.NET feedback

.NET is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

System.Runtime.Serialization.DataContractSerializer class

Article • 01/08/2024

This article provides supplementary remarks to the reference documentation for this API.

Use the [DataContractSerializer](#) class to serialize and deserialize instances of a type into an XML stream or document. For example, you can create a type named `Person` with properties that contain essential data, such as a name and address. You can then create and manipulate an instance of the `Person` class and write all of its property values in an XML document for later retrieval, or in an XML stream for immediate transport. Most important, the [DataContractSerializer](#) is used to serialize and deserialize data sent in Windows Communication Foundation (WCF) messages. Apply the [DataContractAttribute](#) attribute to classes, and the [DataMemberAttribute](#) attribute to class members to specify properties and fields that are serialized.

For a list of types that can be serialized, see [Types Supported by the Data Contract Serializer](#).

To use the [DataContractSerializer](#), first create an instance of a class and an object appropriate to writing or reading the format; for example, an instance of the [XmlDictionaryWriter](#). Then call the [WriteObject](#) method to persist the data. To retrieve data, create an object appropriate to reading the data format (such as an [XmlDictionaryReader](#) for an XML document) and call the [ReadObject](#) method.

For more information about using the [DataContractSerializer](#), see [Serialization and Deserialization](#).

You can set the type of a data contract serializer using the `<dataContractSerializer>` element in a client application configuration file.

Prepare classes for serialization or deserialization

The [DataContractSerializer](#) is used in combination with the [DataContractAttribute](#) and [DataMemberAttribute](#) classes. To prepare a class for serialization, apply the [DataContractAttribute](#) to the class. For each member of the class that returns data that you want to serialize, apply the [DataMemberAttribute](#). You can serialize fields and

properties, regardless of accessibility: private, protected, internal, protected internal, or public.

For example, your schema specifies a `Customer` with an `ID` property, but you already have an existing application that uses a type named `Person` with a `Name` property. To create a type that conforms to the contract, first apply the [DataContractAttribute](#) to the class. Then apply the [DataMemberAttribute](#) to every field or property that you want to serialize.

 **Note**

You can apply the [DataMemberAttribute](#) to both private and public members.

The final format of the XML need not be text. Instead, the [DataContractSerializer](#) writes the data as an XML infoset, which allows you to write the data to any format recognized by the [XmlReader](#) and [XmlWriter](#). It is recommended that you use the [XmlDictionaryReader](#) and [XmlDictionaryWriter](#) classes to read and write, because both are optimized to work with the [DataContractSerializer](#).

If you are creating a class that has fields or properties that must be populated before the serialization or deserialization occurs, use callback attributes, as described in [Version-Tolerant Serialization Callbacks](#).

Add to the collection of known types

When serializing or deserializing an object, it is required that the type is "known" to the [DataContractSerializer](#). Begin by creating an instance of a class that implements `IEnumerable<T>` (such as `List<T>`) and adding the known types to the collection. Then create an instance of the [DataContractSerializer](#) using one of the overloads that takes the `IEnumerable<T>` (for example, `DataContractSerializer(Type, IEnumerable<Type>)`).

 **Note**

Unlike other primitive types, the `DateTimeOffset` structure is not a known type by default, so it must be manually added to the list of known types (see [Data Contract Known Types](#)).

Forward compatibility

The [DataContractSerializer](#) understands data contracts that have been designed to be compatible with future versions of the contract. Such types implement the [IExtensibleDataObject](#) interface. The interface features the [ExtensionData](#) property that returns an [ExtensionDataObject](#) object. For more information, see [Forward-Compatible Data Contracts](#).

Run under partial trust

When instantiating the target object during deserialization, the [DataContractSerializer](#) does not call the constructor of the target object. If you author a [\[DataContract\]](#) type that is accessible from partial trust (that is, it is public and in an assembly that has the [AllowPartiallyTrustedCallers](#) attribute applied) and that performs some security-related actions, you must be aware that the constructor is not called. In particular, the following techniques do not work:

- If you try to restrict partial trust access by making the constructor internal or private, or by adding a [LinkDemand](#) to the constructor -- neither of these have any effect during deserialization under partial trust.
- If you code the class that assumes the constructor has run, the class may get into an invalid internal state that is exploitable.

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

.NET is an open source project.
Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

System.Runtime.Serialization.IExtensibleDataObject interface

Article • 01/08/2024

This article provides supplementary remarks to the reference documentation for this API.

The [IExtensibleDataObject](#) interface provides a single property that sets or returns a structure used to store data that is external to a data contract. The extra data is stored in an instance of the [ExtensionDataObject](#) class and accessed through the [ExtensionData](#) property. In a roundtrip operation where data is received, processed, and sent back, the extra data is sent back to the original sender intact. This is useful to store data received from future versions of the contract. If you do not implement the interface, any extra data is ignored and discarded during a roundtrip operation.

To use this versioning feature

1. Implement the [IExtensibleDataObject](#) interface in a class.
2. Add the [ExtensionData](#) property to your type.
3. Add a private member of type [ExtensionDataObject](#) to the class.
4. Implement get and set methods for the property using the new private member.
5. Apply the [DataContractAttribute](#) attribute to the class. Set the [Name](#) and [Namespace](#) properties to appropriate values if necessary.

For more information about versioning of types, see [Data Contract Versioning](#). For information about creating forward-compatible data contracts, see [Forward-Compatible Data Contracts](#). For more information about data contracts, see [Using Data Contracts](#).

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For

.NET

.NET feedback

.NET is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

more information, see [our contributor guide](#).

System.Runtime.Serialization.XsdDataContractExporter class

Article • 01/08/2024

This article provides supplementary remarks to the reference documentation for this API.

Use the [XsdDataContractExporter](#) class when you have created a Web service that incorporates data represented by common language runtime (CLR) types and when you need to export XML schemas for each type to be consumed by other Web services. That is, [XsdDataContractExporter](#) transforms a set of CLR types into XML schemas. (For more information about the types that can be used, see [Types Supported by the Data Contract Serializer](#).) The schemas can then be exposed through a Web Services Description Language (WSDL) document for use by others who need to interoperate with your service.

Conversely, if you are creating a Web service that must interoperate with an existing Web service, use the [XsdDataContractImporter](#) to transform XML schemas and create the CLR types that represent the data in a selected programming language.

The [XsdDataContractExporter](#) generates an [XmlSchemaSet](#) object that contains the collection of schemas. Access the set of schemas through the [Schemas\(\)](#) property.

ⓘ Note

To quickly generate XML schema definition (XSD) files that other Web services can consume, use the [XsdDataContractExporter](#).

Export schemas into an XmlSchemaSet

To create an instance of the [XmlSchemaSet](#) class that contains XML schema files, you should be aware of the following.

The set of types you are exporting are recorded as an internal set of data contracts. Thus, you can call the [CanExport](#) method multiple times to add new types to the schema set without degrading performance because only the new types will be added to the set. During the [Export](#) operation, the existing schemas are compared to the new schemas being added. If there are conflicts, an exception will be thrown. A conflict is usually

detected if two types with the same data contract name but different contracts (different members) are exported by the same [XsdDataContractExporter](#) instance.

Use the exporter

A recommended way of using this class is as follows:

1. Use one of the [CanExport](#) overloads to determine whether the specified type or set of types can be exported. Use one of the overloads that is appropriate to your requirements.
2. Call the corresponding [Export](#) method.
3. Retrieve the schemas from the [Schemas](#) property.

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

.NET is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

System.CommandLine overview

Article • 04/08/2022

ⓘ Important

`System.CommandLine` is currently in PREVIEW, and this documentation is for version 2.0 beta 4. Some information relates to prerelease product that may be substantially modified before it's released. Microsoft makes no warranties, express or implied, with respect to the information provided here.

The `System.CommandLine` library provides functionality that is commonly needed by command-line apps, such as parsing the command-line input and displaying help text.

Apps that use `System.CommandLine` include the [.NET CLI](#), [additional tools](#), and many [global and local tools](#).

For app developers, the library:

- Lets you focus on writing your app code, since you don't have to write code to parse command-line input or produce a help page.
- Lets you test app code independently of input parsing code.
- Is [trim-friendly](#), making it a good choice for developing a fast, lightweight, AOT-capable CLI app.

Use of the library also benefits app users:

- It ensures that command-line input is parsed consistently according to [POSIX](#) or Windows conventions.
- It automatically supports [tab completion](#) and [response files](#).

NuGet package

The library is available in a NuGet package:

- [System.CommandLine](#)

Next steps

To get started with System.CommandLine, see the following resources:

- [Tutorial: Get started with System.CommandLine](#)

- [Command-line syntax overview](#)

To learn more, see the following resources:

- [How to define commands, options, and arguments](#)
- [How to bind arguments to handlers](#)
- [How to configure dependency injection](#)
- [How to enable and customize tab completion](#)
- [How to customize help](#)
- [How to handle termination](#)
- [How to write middleware and directives](#)
- [System.CommandLine API reference](#)

 [Collaborate with us on GitHub](#)

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

 .NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

Tutorial: Get started with System.CommandLine

Article • 09/21/2022

ⓘ Important

`System.CommandLine` is currently in PREVIEW, and this documentation is for version 2.0 beta 4. Some information relates to prerelease product that may be substantially modified before it's released. Microsoft makes no warranties, express or implied, with respect to the information provided here.

This tutorial shows how to create a .NET command-line app that uses the [System.CommandLine library](#). You'll begin by creating a simple root command that has one option. Then you'll add to that base, creating a more complex app that contains multiple subcommands and different options for each command.

In this tutorial, you learn how to:

- ✓ Create commands, options, and arguments.
- ✓ Specify default values for options.
- ✓ Assign options and arguments to commands.
- ✓ Assign an option recursively to all subcommands under a command.
- ✓ Work with multiple levels of nested subcommands.
- ✓ Create aliases for commands and options.
- ✓ Work with `string`, `string[]`, `int`, `bool`, `FileInfo` and enum option types.
- ✓ Bind option values to command handler code.
- ✓ Use custom code for parsing and validating options.

Prerequisites

- A code editor, such as [Visual Studio Code](#) with the [C# extension](#).
- The [.NET 6 SDK](#).

Or

- [Visual Studio 2022](#) with the [.NET desktop development](#) workload installed.

Create the app

Create a .NET 6 console app project named "scl".

1. Create a folder named *scl* for the project, and then open a command prompt in the new folder.
2. Run the following command:

```
.NET CLI  
dotnet new console --framework net6.0
```

Install the System.CommandLine package

- Run the following command:

```
.NET CLI  
dotnet add package System.CommandLine --prerelease
```

The `--prerelease` option is necessary because the library is still in beta.

1. Replace the contents of *Program.cs* with the following code:

```
C#  
  
using System.CommandLine;  
  
namespace scl;  
  
class Program  
{  
    static async Task<int> Main(string[] args)  
    {  
        var fileOption = new Option<FileInfo?>()  
        {  
            name: "--file",  
            description: "The file to read and display on the  
console."};  
  
        var rootCommand = new RootCommand("Sample app for  
System.CommandLine");  
        rootCommand.AddOption(fileOption);  
  
        rootCommand.SetHandler((file) =>  
        {  
            ReadFile(file!);  
        },  
        fileOption);  
    }  
}
```

```

        return await rootCommand.InvokeAsync(args);
    }

    static void ReadFile(FileInfo file)
    {
        File.ReadLines(file.FullName).ToList()
            .ForEach(line => Console.WriteLine(line));
    }
}

```

The preceding code:

- Creates an [option](#) named `--file` of type [FileInfo](#) and assigns it to the [root command](#):

C#

```

var fileOption = new Option<FileInfo?>(
    name: "--file",
    description: "The file to read and display on the console.");

var rootCommand = new RootCommand("Sample app for System.CommandLine");
rootCommand.AddOption(fileOption);

```

- Specifies that `ReadFile` is the method that will be called when the root command is invoked:

C#

```

rootCommand.SetHandler((file) =>
{
    ReadFile(file!);
},
fileOption);

```

- Displays the contents of the specified file when the root command is invoked:

C#

```

static void ReadFile(FileInfo file)
{
    File.ReadLines(file.FullName).ToList()
        .ForEach(line => Console.WriteLine(line));
}

```

Test the app

You can use any of the following ways to test while developing a command-line app:

- Run the `dotnet build` command, and then open a command prompt in the `scl/bin/Debug/net6.0` folder to run the executable:

Console

```
dotnet build  
cd bin/Debug/net6.0  
scl --file scl.runtimeconfig.json
```

- Use `dotnet run` and pass option values to the app instead of to the `run` command by including them after `--`, as in the following example:

.NET CLI

```
dotnet run -- --file scl.runtimeconfig.json
```

In .NET 7.0.100 SDK Preview, you can use the `commandLineArgs` of a `launchSettings.json` file by running the command `dotnet run --launch-profile <profilename>`.

- [Publish the project to a folder](#), open a command prompt to that folder, and run the executable:

Console

```
dotnet publish -o publish  
cd ./publish  
scl --file scl.runtimeconfig.json
```

- In Visual Studio 2022, select **Debug > Debug Properties** from the menu, and enter the options and arguments in the **Command line arguments** box. For example:



Then run the app, for example by pressing **Ctrl+F5**.

This tutorial assumes you're using the first of these options.

When you run the app, it displays the contents of the file specified by the `--file` option.

```
Output

{
  "runtimeOptions": {
    "tfm": "net6.0",
    "framework": {
      "name": "Microsoft.NETCore.App",
      "version": "6.0.0"
    }
  }
}
```

Help output

`System.CommandLine` automatically provides help output:

```
Console

scl --help
```

```
Output

Description:
  Sample app for System.CommandLine

Usage:
  scl [options]

Options:
  --file <file>    The file to read and display on the console.
  --version         Show version information
  -?, -h, --help   Show help and usage information
```

Version output

`System.CommandLine` automatically provides version output:

```
Console

scl --version
```

```
Output
```

Add a subcommand and options

In this section, you:

- Create more options.
- Create a subcommand.
- Assign the new options to the new subcommand.

The new options will let you configure the foreground and background text colors and the readout speed. These features will be used to read a collection of quotes that comes from the [Teleprompter console app tutorial](#).

1. Copy the [sampleQuotes.txt](#) file from the GitHub repository for this sample into your project directory. For information on how to download files, see the instructions in [Samples and Tutorials](#).
2. Open the project file and add an `<ItemGroup>` element just before the closing `</Project>` tag:

XML

```
<ItemGroup>
  <Content Include="sampleQuotes.txt">
    <CopyToOutputDirectory>Always</CopyToOutputDirectory>
  </Content>
</ItemGroup>
```

Adding this markup causes the text file to be copied to the `bin/debug/net6.0` folder when you build the app. So when you run the executable in that folder, you can access the file by name without specifying a folder path.

3. In `Program.cs`, after the code that creates the `--file` option, create options to control the readout speed and text colors:

C#

```
var delayOption = new Option<int>(
  name: "--delay",
  description: "Delay between lines, specified as milliseconds per character in a line.",
  getDefaultValue: () => 42);
```

```
var fgcolorOption = new Option<ConsoleColor>(
    name: "--fgcolor",
    description: "Foreground color of text displayed on the console.",
    getDefaultValue: () => ConsoleColor.White);

var lightModeOption = new Option<bool>(
    name: "--light-mode",
    description: "Background color of text displayed on the console:
default is black, light mode is white.");
```

4. After the line that creates the root command, delete the line that adds the `--file` option to it. You're removing it here because you'll add it to a new subcommand.

C#

```
var rootCommand = new RootCommand("Sample app for System.CommandLine");
//rootCommand.AddOption(fileOption);
```

5. After the line that creates the root command, create a `read` subcommand. Add the options to this subcommand, and add the subcommand to the root command.

C#

```
var readCommand = new Command("read", "Read and display the file.")
{
    fileOption,
    delayOption,
    fgcolorOption,
    lightModeOption
};
rootCommand.AddCommand(readCommand);
```

6. Replace the `SetHandler` code with the following `SetHandler` code for the new subcommand:

C#

```
readCommand.SetHandler(async (file, delay, fgcolor, lightMode) =>
{
    await ReadFile(file!, delay, fgcolor, lightMode);
},
fileOption, delayOption, fgcolorOption, lightModeOption);
```

You're no longer calling `SetHandler` on the root command because the root command no longer needs a handler. When a command has subcommands, you

typically have to specify one of the subcommands when invoking a command-line app.

7. Replace the `ReadFile` handler method with the following code:

```
C#  
  
internal static async Task ReadFile(  
    FileInfo file, int delay, ConsoleColor fgColor, bool lightMode)  
{  
    Console.BackgroundColor = lightMode ? ConsoleColor.White :  
    ConsoleColor.Black;  
    Console.ForegroundColor = fgColor;  
    List<string> lines = File.ReadLines(file.FullName).ToList();  
    foreach (string line in lines)  
    {  
        Console.WriteLine(line);  
        await Task.Delay(delay * line.Length);  
    };  
}
```

The app now looks like this:

```
C#  
  
using System.CommandLine;  
  
namespace scl;  
  
class Program  
{  
    static int Main(string[] args)  
    {  
        var fileOption = new Option<FileInfo?>(  
            name: "--file",  
            description: "The file to read and display on the console.");  
  
        var delayOption = new Option<int>(  
            name: "--delay",  
            description: "Delay between lines, specified as milliseconds per  
character in a line.",  
            getDefaultValue: () => 42);  
  
        var fgcolorOption = new Option<ConsoleColor>(  
            name: "--fgcolor",  
            description: "Foreground color of text displayed on the  
console.",  
            getDefaultValue: () => ConsoleColor.White);  
  
        var lightModeOption = new Option<bool>(  
            name: "--light-mode",  
            description: "Background color of text displayed on the console:");  
    }  
}
```

```

default is black, light mode is white.");

    var rootCommand = new RootCommand("Sample app for
System.CommandLine");
    //rootCommand.AddOption(fileOption);

    var readCommand = new Command("read", "Read and display the file.")
    {
        fileOption,
        delayOption,
        fgcolorOption,
        lightModeOption
    };
    rootCommand.AddCommand(readCommand);

    readCommand.SetHandler(async (file, delay, fgcolor, lightMode) =>
    {
        await ReadFile(file!, delay, fgcolor, lightMode);
    },
    fileOption, delayOption, fgcolorOption, lightModeOption);

    return rootCommand.InvokeAsync(args).Result;
}

internal static async Task ReadFile(
    FileInfo file, int delay, ConsoleColor fgColor, bool lightMode)
{
    ConsoleColor.BackgroundColor = lightMode ? ConsoleColor.White :
ConsoleColor.Black;
    ConsoleColor.ForegroundColor = fgColor;
    List<string> lines = File.ReadLines(file.FullName).ToList();
    foreach (string line in lines)
    {
        Console.WriteLine(line);
        await Task.Delay(delay * line.Length);
    };
}
}

```

Test the new subcommand

Now if you try to run the app without specifying the subcommand, you get an error message followed by a help message that specifies the subcommand that is available.

Console

```
scl --file sampleQuotes.txt
```

Output

```
'--file' was not matched. Did you mean one of the following?
--help
Required command was not provided.
Unrecognized command or argument '--file'.
Unrecognized command or argument 'sampleQuotes.txt'.

Description:
    Sample app for System.CommandLine

Usage:
    scl [command] [options]

Options:
    --version      Show version information
    -?, -h, --help Show help and usage information

Commands:
    read  Read and display the file.
```

The help text for subcommand `read` shows that four options are available. It shows valid values for the enum.

Console

```
scl read -h
```

Output

```
Description:
    Read and display the file.

Usage:
    scl read [options]

Options:
    --file <file>                                         The file to
    read and display on the console.
    --delay <delay>                                         Delay between
    lines, specified as milliseconds per
    character in a
    line. [default: 42]
    --fgcolor                                              Foreground
    color of text displayed on the console.
    <Black|Blue|Cyan|DarkBlue|DarkCyan|DarkGray|DarkGreen|Dark
    White|Magenta|DarkRed|DarkYellow|Gray|Green|Magenta|Red|White|Ye
    llow>                                                 [default:
    --light-mode                                           Background
    color of text displayed on the console:
    default is
```

```
black, light mode is white.  
-?, -h, --help  
usage information
```

Show help and

Run subcommand `read` specifying only the `--file` option, and you get the default values for the other three options.

Console

```
scl read --file sampleQuotes.txt
```

The 42 milliseconds per character default delay causes a slow readout speed. You can speed it up by setting `--delay` to a lower number.

Console

```
scl read --file sampleQuotes.txt --delay 0
```

You can use `--fgcolor` and `--light-mode` to set text colors:

Console

```
scl read --file sampleQuotes.txt --fgcolor red --light-mode
```

Provide an invalid value for `--delay` and you get an error message:

Console

```
scl read --file sampleQuotes.txt --delay forty-two
```

Output

```
Cannot parse argument 'forty-two' for option '--int' as expected type  
'System.Int32'.
```

Provide an invalid value for `--file` and you get an exception:

Console

```
scl read --file nofile
```

Output

```
Unhandled exception: System.IO.FileNotFoundException:  
Could not find file 'C:\bin\Debug\net6.0\nofile'.
```

Add subcommands and custom validation

This section creates the final version of the app. When finished, the app will have the following commands and options:

- root command with a global* option named `--file`
 - `quotes` command
 - `read` command with options named `--delay`, `--fgcolor`, and `--light-mode`
 - `add` command with arguments named `quote` and `byline`
 - `delete` command with option named `--search-terms`

* A global option is available to the command it's assigned to and recursively to all its subcommands.

Here's sample command line input that invokes each of the available commands with its options and arguments:

Console

```
scl quotes read --file sampleQuotes.txt --delay 40 --fgcolor red --light-mode  
scl quotes add "Hello world!" "Nancy Davolio"  
scl quotes delete --search-terms David "You can do" Antoine "Perfection is  
achieved"
```

1. In `Program.cs`, replace the code that creates the `--file` option with the following code:

C#

```
var fileOption = new Option<FileInfo?>()  
    name: "--file",  
    description: "An option whose argument is parsed as a FileInfo",  
    isDefault: true,  
    parseArgument: result =>  
    {  
        if (result.Tokens.Count == 0)  
        {  
            return new FileInfo("sampleQuotes.txt");  
  
        }  
        string? filePath = result.Tokens.Single().Value;
```

```
        if (!File.Exists(filePath))
    {
        result.ErrorMessage = "File does not exist";
        return null;
    }
    else
    {
        return new FileInfo(filePath);
    }
});
```

This code uses `ParseArgument<T>` to provide custom parsing, validation, and error handling.

Without this code, missing files are reported with an exception and stack trace.
With this code just the specified error message is displayed.

This code also specifies a default value, which is why it sets `isDefault` to `true`. If you don't set `isDefault` to `true`, the `parseArgument` delegate doesn't get called when no input is provided for `--file`.

2. After the code that creates `lightModeOption`, add options and arguments for the `add` and `delete` commands:

```
C#  
  
var searchTermsOption = new Option<string[]>(
    name: "--search-terms",
    description: "Strings to search for when deleting entries.")
{ IsRequired = true, AllowMultipleArgumentsPerToken = true };  
  
var quoteArgument = new Argument<string>(
    name: "quote",
    description: "Text of quote.");  
  
var bylineArgument = new Argument<string>(
    name: "byline",
    description: "Byline of quote.");
```

The `AllowMultipleArgumentsPerToken` setting lets you omit the `--search-terms` option name when specifying elements in the list after the first one. It makes the following examples of command-line input equivalent:

Console

```
scl quotes delete --search-terms David "You can do"  
scl quotes delete --search-terms David --search-terms "You can do"
```

3. Replace the code that creates the root command and the `read` command with the following code:

```
C#  
  
var rootCommand = new RootCommand("Sample app for System.CommandLine");  
rootCommand.AddGlobalOption(fileOption);  
  
var quotesCommand = new Command("quotes", "Work with a file that  
contains quotes.");  
rootCommand.AddCommand(quotesCommand);  
  
var readCommand = new Command("read", "Read and display the file.")  
{  
    delayOption,  
    fgcolorOption,  
    lightModeOption  
};  
quotesCommand.AddCommand(readCommand);  
  
var deleteCommand = new Command("delete", "Delete lines from the  
file.");  
deleteCommand.AddOption(searchTermsOption);  
quotesCommand.AddCommand(deleteCommand);  
  
var addCommand = new Command("add", "Add an entry to the file.");  
addCommand.AddArgument(quoteArgument);  
addCommand.AddArgument(bylineArgument);  
addCommand.AddAlias("insert");  
quotesCommand.AddCommand(addCommand);
```

This code makes the following changes:

- Removes the `--file` option from the `read` command.
- Adds the `--file` option as a global option to the root command.
- Creates a `quotes` command and adds it to the root command.
- Adds the `read` command to the `quotes` command instead of to the root command.
- Creates `add` and `delete` commands and adds them to the `quotes` command.

The result is the following command hierarchy:

- Root command
 - `quotes`
 - `read`

- o `add`
- o `delete`

The app now implements the recommended pattern where the parent command (`quotes`) specifies an area or group, and its children commands (`read`, `add`, `delete`) are actions.

Global options are applied to the command and recursively to subcommands. Since `--file` is on the root command, it will be available automatically in all subcommands of the app.

4. After the `SetHandler` code, add new `SetHandler` code for the new subcommands:

```
C#
deleteCommand.SetHandler((file, searchTerms) =>
{
    DeleteFromFile(file!, searchTerms);
},
fileOption, searchTermsOption);

addCommand.SetHandler((file, quote, byline) =>
{
    AddToFile(file!, quote, byline);
},
fileOption, quoteArgument, bylineArgument);
```

Subcommand `quotes` doesn't have a handler because it isn't a leaf command. Subcommands `read`, `add`, and `delete` are leaf commands under `quotes`, and `SetHandler` is called for each of them.

5. Add the handlers for `add` and `delete`.

```
C#
internal static void DeleteFromFile(FileInfo file, string[] searchTerms)
{
    Console.WriteLine("Deleting from file");
    File.WriteAllLines(
        file.FullName, File.ReadLines(file.FullName)
            .Where(line => searchTerms.All(s =>
                !line.Contains(s))).ToList());
}

internal static void AddToFile(FileInfo file, string quote, string byline)
{
    Console.WriteLine("Adding to file");
```

```

        using StreamWriter? writer = file.AppendText();
        writer.WriteLine(${Environment.NewLine}${Environment.NewLine}
{quote}");
        writer.WriteLine(${Environment.NewLine}-{byline}");
        writer.Flush();
    }

```

The finished app looks like this:

C#

```

using System.CommandLine;

namespace scl;

class Program
{
    static async Task<int> Main(string[] args)
    {
        var fileOption = new Option<FileInfo?>(
            name: "--file",
            description: "An option whose argument is parsed as a FileInfo",
            isDefault: true,
            parseArgument: result =>
        {
            if (result.Tokens.Count == 0)
            {
                return new FileInfo("sampleQuotes.txt");
            }

            string? filePath = result.Tokens.Single().Value;
            if (!File.Exists(filePath))
            {
                result.ErrorMessage = "File does not exist";
                return null;
            }
            else
            {
                return new FileInfo(filePath);
            }
        });
    }

    var delayOption = new Option<int>(
        name: "--delay",
        description: "Delay between lines, specified as milliseconds per
character in a line.",
        getDefaultValue: () => 42);

    var fgcolorOption = new Option<ConsoleColor>(
        name: "--fgcolor",
        description: "Foreground color of text displayed on the
console.",
        getDefaultValue: () => ConsoleColor.White);

```

```
var lightModeOption = new Option<bool>(
    name: "--light-mode",
    description: "Background color of text displayed on the console:
default is black, light mode is white.");

var searchTermsOption = new Option<string[]>(
    name: "--search-terms",
    description: "Strings to search for when deleting entries.")
{ IsRequired = true, AllowMultipleArgumentsPerToken = true };

var quoteArgument = new Argument<string>(
    name: "quote",
    description: "Text of quote.");

var bylineArgument = new Argument<string>(
    name: "byline",
    description: "Byline of quote.");

var rootCommand = new RootCommand("Sample app for
System.CommandLine");
rootCommand.AddGlobalOption(fileOption);

var quotesCommand = new Command("quotes", "Work with a file that
contains quotes.");
rootCommand.AddCommand(quotesCommand);

var readCommand = new Command("read", "Read and display the file.")
{
    delayOption,
    fgcolorOption,
    lightModeOption
};
quotesCommand.AddCommand(readCommand);

var deleteCommand = new Command("delete", "Delete lines from the
file.");
deleteCommand.AddOption(searchTermsOption);
quotesCommand.AddCommand(deleteCommand);

var addCommand = new Command("add", "Add an entry to the file.");
addCommand.AddArgument(quoteArgument);
addCommand.AddArgument(bylineArgument);
addCommand.AddAlias("insert");
quotesCommand.AddCommand(addCommand);

readCommand.SetHandler(async (file, delay, fgcolor, lightMode) =>
{
    await ReadFile(file!, delay, fgcolor, lightMode);
},
fileOption, delayOption, fgcolorOption, lightModeOption);

deleteCommand.SetHandler((file, searchTerms) =>
{
    DeleteFromFile(file!, searchTerms);
```

```

        },
        fileOption, searchTermsOption);

    addCommand.SetHandler((file, quote, byline) =>
    {
        AddToFile(file!, quote, byline);
    },
    fileOption, quoteArgument, bylineArgument);

    return await rootCommand.InvokeAsync(args);
}

internal static async Task ReadFile(
    FileInfo file, int delay, ConsoleColor fgColor, bool
lightMode)
{
    Console.BackgroundColor = lightMode ? ConsoleColor.White :
ConsoleColor.Black;
    Console.ForegroundColor = fgColor;
    var lines = File.ReadLines(file.FullName).ToList();
    foreach (string line in lines)
    {
        Console.WriteLine(line);
        await Task.Delay(delay * line.Length);
    };
}

internal static void DeleteFromFile(FileInfo file, string[] searchTerms)
{
    Console.WriteLine("Deleting from file");
    File.WriteAllLines(
        file.FullName, File.ReadLines(file.FullName)
        .Where(line => searchTerms.All(s =>
!line.Contains(s))).ToList());
}

internal static void AddToFile(FileInfo file, string quote, string
byline)
{
    Console.WriteLine("Adding to file");
    using StreamWriter? writer = file.AppendText();
    writer.WriteLine($"{Environment.NewLine}{Environment.NewLine}
{quote}");
    writer.WriteLine($"{Environment.NewLine}-{byline}");
    writer.Flush();
}
}

```

Build the project, and then try the following commands.

Submit a nonexistent file to `--file` with the `read` command, and you get an error message instead of an exception and stack trace:

Console

```
scl quotes read --file myfile
```

Output

```
File does not exist
```

Try to run subcommand `quotes` and you get a message directing you to use `read`, `add`, or `delete`:

Console

```
scl quotes
```

Output

```
Required command was not provided.
```

Description:

```
Work with a file that contains quotes.
```

Usage:

```
scl quotes [command] [options]
```

Options:

```
--file <file> An option whose argument is parsed as a FileInfo [default:  
sampleQuotes.txt]  
-?, -h, --help Show help and usage information
```

Commands:

| | |
|------------------------------|-----------------------------|
| read | Read and display the file. |
| delete | Delete lines from the file. |
| add, insert <quote> <byline> | Add an entry to the file. |

Run subcommand `add`, and then look at the end of the text file to see the added text:

Console

```
scl quotes add "Hello world!" "Nancy Davolio"
```

Run subcommand `delete` with search strings from the beginning of the file, and then look at the beginning of the text file to see where text was removed:

Console

```
scl quotes delete --search-terms David "You can do" Antoine "Perfection is achieved"
```

ⓘ Note

If you're running in the `bin/debug/net6.0` folder, that folder is where you'll find the file with changes from the `add` and `delete` commands. The copy of the file in the project folder remains unchanged.

Next steps

In this tutorial, you created a simple command-line app that uses `System.CommandLine`. To learn more about the library, see [System.CommandLine overview](#).

⌚ Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

[⌚ Open a documentation issue](#)

[👤 Provide product feedback](#)

Command-line syntax overview for System.CommandLine

Article • 09/21/2022

ⓘ Important

`System.CommandLine` is currently in PREVIEW, and this documentation is for version 2.0 beta 4. Some information relates to prerelease product that may be substantially modified before it's released. Microsoft makes no warranties, express or implied, with respect to the information provided here.

This article explains the command-line syntax that `System.CommandLine` recognizes. The information will be useful to users as well as developers of .NET command-line apps, including the [.NET CLI](#).

Tokens

`System.CommandLine` parses command-line input into *tokens*, which are strings delimited by spaces. For example, consider the following command line:

.NET CLI

```
dotnet tool install dotnet-suggest --global --verbosity quiet
```

This input is parsed by the `dotnet` application into tokens `tool`, `install`, `dotnet-suggest`, `--global`, `--verbosity`, and `quiet`.

Tokens are interpreted as commands, options, or arguments. The command-line app that is being invoked determines how the tokens after the first one are interpreted. The following table shows how `System.CommandLine` interprets the preceding example:

[+] Expand table

| Token | Parsed as |
|-----------------------------|------------------------------|
| <code>tool</code> | Subcommand |
| <code>install</code> | Subcommand |
| <code>dotnet-suggest</code> | Argument for install command |

| Token | Parsed as |
|-------------|---------------------------------|
| --global | Option for install command |
| --verbosity | Option for install command |
| quiet | Argument for --verbosity option |

A token can contain spaces if it's enclosed in quotation marks (""). Here's an example:

```
Console  
dotnet tool search "ef migrations add"
```

Commands

A *command* in command-line input is a token that specifies an action or defines a group of related actions. For example:

- In `dotnet run`, `run` is a command that specifies an action.
- In `dotnet tool install`, `install` is a command that specifies an action, and `tool` is a command that specifies a group of related commands. There are other tool-related commands, such as `tool uninstall`, `tool list`, and `tool update`.

Root commands

The *root command* is the one that specifies the name of the app's executable. For example, the `dotnet` command specifies the `dotnet.exe` executable.

Subcommands

Most command-line apps support *subcommands*, also known as *verbs*. For example, the `dotnet` command has a `run` subcommand that you invoke by entering `dotnet run`.

Subcommands can have their own subcommands. In `dotnet tool install`, `install` is a subcommand of `tool`.

Options

An option is a named parameter that can be passed to a command. [POSIX ↗](#) CLIs typically prefix the option name with two hyphens (--). The following example shows

two options:

.NET CLI

```
dotnet tool update dotnet-suggest --verbosity quiet --global
^-----^          ^-----^
```

As this example illustrates, the value of the option may be explicit (`quiet` for `--verbosity`) or implicit (nothing follows `--global`). Options that have no value specified are typically Boolean parameters that default to `true` if the option is specified on the command line.

For some Windows command-line apps, you identify an option by using a leading slash (/) with the option name. For example:

Console

```
msbuild /version
^-----^
```

`System.CommandLine` supports both POSIX and Windows prefix conventions. When you [configure an option](#), you specify the option name including the prefix.

Arguments

An argument is a value passed to an option or a command. The following examples show an argument for the `verbosity` option and an argument for the `build` command.

Console

```
dotnet tool update dotnet-suggest --verbosity quiet --global
^---^
```

Console

```
dotnet build myapp.csproj
^-----^
```

Arguments can have default values that apply if no argument is explicitly provided. For example, many options are implicitly Boolean parameters with a default of `true` when the option name is in the command line. The following command-line examples are equivalent:

.NET CLI

```
dotnet tool update dotnet-suggest --global  
dotnet tool update dotnet-suggest --global true
```

Some options have required arguments. For example in the .NET CLI, `--output` requires a folder name argument. If the argument is not provided, the command fails.

Arguments can have expected types, and `System.CommandLine` displays an error message if an argument can't be parsed into the expected type. For example, the following command errors because "silent" isn't one of the valid values for `--verbosity`:

.NET CLI

```
dotnet build --verbosity silent
```

Output

```
Cannot parse argument 'silent' for option '-v' as expected type  
'Microsoft.DotNet.Cli.VerbosityOptions'. Did you mean one of the following?  
Detailed  
Diagnostic  
Minimal  
Normal  
Quiet
```

Arguments also have expectations about how many values can be provided. Examples are provided in the [section on argument arity](#).

Order of options and arguments

You can provide options before arguments or arguments before options on the command line. The following commands are equivalent:

.NET CLI

```
dotnet add package System.CommandLine --prerelease  
dotnet add package --prerelease System.CommandLine
```

Options can be specified in any order. The following commands are equivalent:

.NET CLI

```
dotnet add package System.CommandLine --prerelease --no-restore --source https://api.nuget.org/v3/index.json  
dotnet add package System.CommandLine --source https://api.nuget.org/v3/index.json --no-restore --prerelease
```

When there are multiple arguments, the order does matter. The following commands are not necessarily equivalent:

Console

```
myapp argument1 argument2  
myapp argument2 argument1
```

These commands pass a list with the same values to the command handler code, but they differ in the order of the values, which could lead to different results.

Aliases

In both POSIX and Windows, it's common for some commands and options to have aliases. These are usually short forms that are easier to type. Aliases can also be used for other purposes, such as to [simulate case-insensitivity](#) and to [support alternate spellings of a word](#).

POSIX short forms typically have a single leading hyphen followed by a single character. The following commands are equivalent:

.NET CLI

```
dotnet build --verbosity quiet  
dotnet build -v quiet
```

The [GNU standard](#) recommends automatic aliases. That is, you can enter any part of a long-form command or option name and it will be accepted. This behavior would make the following command lines equivalent:

.NET CLI

```
dotnet publish --output ./publish  
dotnet publish --outpu ./publish  
dotnet publish --outp ./publish  
dotnet publish --out ./publish
```

```
dotnet publish --ou ./publish  
dotnet publish --o ./publish
```

`System.CommandLine` doesn't support automatic aliases.

Case sensitivity

Command and option names and aliases are case-sensitive by default according to POSIX convention, and `System.CommandLine` follows this convention. If you want your CLI to be case insensitive, define aliases for the various casing alternatives. For example, `--additional-probing-path` could have aliases `--Additional-Probing-Path` and `--ADDITIONAL-PROBING-PATH`.

In some command-line tools, a difference in casing specifies a difference in function. For example, `git clean -X` behaves differently than `git clean -x`. The .NET CLI is all lowercase.

Case sensitivity does not apply to argument values for options that are based on enums. Enum names are matched regardless of casing.

The `--` token

POSIX convention interprets the double-dash (`--`) token as an escape mechanism. Everything that follows the double-dash token is interpreted as arguments for the command. This functionality can be used to submit arguments that look like options, since it prevents them from being interpreted as options.

Suppose `myapp` takes a `message` argument, and you want the value of `message` to be `--interactive`. The following command line might give unexpected results.

Console

```
myapp --interactive
```

If `myapp` doesn't have an `--interactive` option, the `--interactive` token is interpreted as an argument. But if the app does have an `--interactive` option, this input will be interpreted as referring to that option.

The following command line uses the double-dash token to set the value of the `message` argument to "`--interactive`":

Console

```
myapp -- --interactive  
^^
```

`System.CommandLine` supports this double-dash functionality.

Option-argument delimiters

`System.CommandLine` lets you use a space, '=' or ':' as the delimiter between an option name and its argument. For example, the following commands are equivalent:

.NET CLI

```
dotnet build -v quiet  
dotnet build -v=quiet  
dotnet build -v:quiet
```

A POSIX convention lets you omit the delimiter when you are specifying a single-character option alias. For example, the following commands are equivalent:

Console

```
myapp -vquiet  
myapp -v quiet
```

`System.CommandLine` supports this syntax by default.

Argument arity

The *arity* of an option or command's argument is the number of values that can be passed if that option or command is specified.

Arity is expressed with a minimum value and a maximum value, as the following table illustrates:

[] [Expand table](#)

| Min | Max | Example validity | Example |
|-----|-----|---------------------------|---------------|
| 0 | 0 | Valid: --file | --file |
| | | Invalid: --file a.json | --file a.json |

| Min | Max | Example validity | Example |
|-----|-----|------------------|-----------------------------|
| | | Invalid: | --file a.json --file b.json |
| 0 | 1 | Valid: | --flag |
| | | Valid: | --flag true |
| | | Valid: | --flag false |
| | | Invalid: | --flag false --flag false |
| 1 | 1 | Valid: | --file a.json |
| | | Invalid: | --file |
| | | Invalid: | --file a.json --file b.json |
| 0 | n | Valid: | --file |
| | | Valid: | --file a.json |
| | | Valid: | --file a.json --file b.json |
| 1 | n | Valid: | --file a.json |
| | | Valid: | --file a.json b.json |
| | | Invalid: | --file |

`System.CommandLine` has an [ArgumentArity](#) struct for defining arity, with the following values:

- [Zero](#) - No values allowed.
- [ZeroOrOne](#) - May have one value, may have no values.
- [ExactlyOne](#) - Must have one value.
- [ZeroOrMore](#) - May have one value, multiple values, or no values.
- [OneOrMore](#) - May have multiple values, must have at least one value.

Arity can often be inferred from the type. For example, an `int` option has arity of `ExactlyOne`, and a `List<int>` option has arity `OneOrMore`.

Option overrides

If the arity maximum is 1, `System.CommandLine` can still be configured to accept multiple instances of an option. In that case, the last instance of a repeated option overwrites any earlier instances. In the following example, the value 2 would be passed to the `myapp` command.

Console

```
myapp --delay 3 --message example --delay 2
```

Multiple arguments

If the arity maximum is more than one, `System.CommandLine` can be configured to accept multiple arguments for one option without repeating the option name.

In the following example, the list passed to the `myapp` command would contain "a", "b", "c", and "d":

Console

```
myapp --list a b c --list d
```

Option bundling

POSIX recommends that you support *bundling* of single-character options, also known as *stacking*. Bundled options are single-character option aliases specified together after a single hyphen prefix. Only the last option can specify an argument. For example, the following command lines are equivalent:

Console

```
git clean -f -d -x  
git clean -fdx
```

If an argument is provided after an option bundle, it applies to the last option in the bundle. The following command lines are equivalent:

Console

```
myapp -a -b -c arg  
myapp -abc arg
```

In both variants in this example, the argument `arg` would apply only to the option `-c`.

Boolean options (flags)

If `true` or `false` is passed for an option having a `bool` argument, it's parsed as expected. But an option whose argument type is `bool` typically doesn't require an argument to be specified. Boolean options, sometimes called "flags", typically have an arity of [ZeroOrOne](#). The presence of the option name on the command line, with no argument following it, results in a default value of `true`. The absence of the option name in command-line input results in a value of `false`. If the `myapp` command prints out the value of a Boolean option named `--interactive`, the following input creates the following output:

```
Console
myapp
myapp --interactive
myapp --interactive false
myapp --interactive true

Output
False
True
False
True
```

The `--help` option

Command-line apps typically provide an option to display a brief description of the available commands, options, and arguments. `System.CommandLine` automatically generates help output. For example:

```
.NET CLI
dotnet list --help

Output
Description:
List references or packages of a .NET project.

Usage:
dotnet [options] list [<PROJECT | SOLUTION>] [command]

Arguments:
<PROJECT | SOLUTION> The project or solution file to operate on. If a
file is not specified, the command will search the current directory for
```

```
one.
```

Options:

```
-?, -h, --help Show command line help.
```

Commands:

```
package List all package references of the project or solution.  
reference List all project-to-project references of the project.
```

App users might be accustomed to different ways to request help on different platforms, so apps built on `System.CommandLine` respond to many ways of requesting help. The following commands are all equivalent:

.NET CLI

```
dotnet --help  
dotnet -h  
dotnet /h  
dotnet -?  
dotnet /?
```

Help output doesn't necessarily show all available commands, arguments, and options. Some of them may be *hidden*, which means they don't show up in help output but they can be specified on the command line.

The `--version` option

Apps built on `System.CommandLine` automatically provide the version number in response to the `--version` option used with the root command. For example:

.NET CLI

```
dotnet --version
```

Output

```
6.0.100
```

Response files

A *response file* is a file that contains a set of `tokens` for a command-line app. Response files are a feature of `System.CommandLine` that is useful in two scenarios:

- To invoke a command-line app by specifying input that is longer than the character limit of the terminal.
- To invoke the same command repeatedly without retyping the whole line.

To use a response file, enter the file name prefixed by an `@` sign wherever in the line you want to insert commands, options, and arguments. The `.rsp` file extension is a common convention, but you can use any file extension.

The following lines are equivalent:

.NET CLI

```
dotnet build --no-restore --output ./build-output/
dotnet @sample1.rsp
dotnet build @sample2.rsp --output ./build-output/
```

Contents of `sample1.rsp`:

Console

```
build
--no-restore
--output
./build-output/
```

Contents of `sample2.rsp`:

Console

```
--no-restore
```

Here are syntax rules that determine how the text in a response file is interpreted:

- Tokens are delimited by spaces. A line that contains *Good morning!* is treated as two tokens, *Good* and *morning!*.
- Multiple tokens enclosed in quotes are interpreted as a single token. A line that contains "Good morning!" is treated as one token, *Good morning!*.
- Any text between a `#` symbol and the end of the line is treated as a comment and ignored.
- Tokens prefixed with `@` can reference additional response files.
- The response file can have multiple lines of text. The lines are concatenated and interpreted as a sequence of tokens.

Directives

`System.CommandLine` introduces a syntactic element called a *directive*. The `[parse]` directive is an example. When you include `[parse]` after the app's name, `System.CommandLine` displays a diagram of the parse result instead of invoking the command-line app:

.NET CLI

```
dotnet [parse] build --no-restore --output ./build-output/  
      ^-----^
```

Output

```
[ dotnet [ build [ --no-restore <True> ] [ --output <./build-output/> ] ] ]
```

The purpose of directives is to provide cross-cutting functionality that can apply across command-line apps. Because directives are syntactically distinct from the app's own syntax, they can provide functionality that applies across apps.

A directive must conform to the following syntax rules:

- It's a token on the command line that comes after the app's name but before any subcommands or options.
- It's enclosed in square brackets.
- It doesn't contain spaces.

An unrecognized directive is ignored without causing a parsing error.

A directive can include an argument, separated from the directive name by a colon.

The following directives are built in:

- `[parse]`
- `[suggest]`

The `[parse]` directive

Both users and developers may find it useful to see how an app will interpret a given input. One of the default features of a `System.CommandLine` app is the `[parse]` directive, which lets you preview the result of parsing command input. For example:

Console

```
myapp [parse] --delay not-an-int --interactive --file filename.txt extra
```

Output

```
![ myapp [ --delay !<not-an-int> ] [ --interactive <True> ] [ --file <filename.txt> ] *[ --fgcolor <White> ] ] ???--> extra
```

In the preceding example:

- The command (`myapp`), its child options, and the arguments to those options are grouped using square brackets.
- For the option result `[--delay !<not-an-int>]`, the `!` indicates a parsing error. The value `not-an-int` for an `int` option can't be parsed to the expected type. The error is also flagged by `!` in front of the command that contains the errored option: `![myapp....`.
- For the option result `*[--fgcolor <White>]`, the option wasn't specified on the command line, so the configured default was used. `white` is the effective value for this option. The asterisk indicates that the value is the default.
- `???-->` points to input that wasn't matched to any of the app's commands or options.

The `[suggest]` directive

The `[suggest]` directive lets you search for commands when you don't know the exact command.

.NET CLI

```
dotnet [suggest] buil
```

Output

```
build  
build-server  
msbuild
```

Design guidance

The following sections present guidance that we recommend you follow when designing a CLI. Think of what your app expects on the command line as similar to what a REST

API server expects in the URL. Consistent rules for REST APIs are what make them readily usable to client app developers. In the same way, users of your command-line apps will have a better experience if the CLI design follows common patterns.

Once you create a CLI it is hard to change, especially if your users have used your CLI in scripts they expect to keep running. The guidelines here were developed after the .NET CLI, and it doesn't always follow these guidelines. We are updating the .NET CLI where we can do it without introducing breaking changes. An example of this work is the new design for `dotnet new` in .NET 7.

Commands and subcommands

If a command has subcommands, the command should function as an area, or a grouping identifier for the subcommands, rather than specify an action. When you invoke the app, you specify the grouping command and one of its subcommands. For example, try to run `dotnet tool`, and you get an error message because the `tool` command only identifies a group of tool-related subcommands, such as `install` and `list`. You can run `dotnet tool install`, but `dotnet tool` by itself would be incomplete.

One of the ways that defining areas helps your users is that it organizes the help output.

Within a CLI there is often an implicit area. For example, in the .NET CLI, the implicit area is the project and in the Docker CLI it is the image. As a result, you can use `dotnet build` without including an area. Consider whether your CLI has an implicit area. If it does, consider whether to allow the user to optionally include or omit it as in `docker build` and `docker image build`. If you optionally allow the implicit area to be typed by your user, you also automatically have help and tab completion for this grouping of commands. Supply the optional use of the implicit group by defining two commands that perform the same operation.

Options as parameters

Options should provide parameters to commands, rather than specifying actions themselves. This is a recommended design principle although it isn't always followed by `System.CommandLine` (`--help` displays help information).

Short-form aliases

In general, we recommend that you minimize the number of short-form option aliases that you define.

In particular, avoid using any of the following aliases differently than their common usage in the .NET CLI and other .NET command-line apps:

- `-i` for `--interactive`.

This option signals to the user that they may be prompted for inputs to questions that the command needs answered. For example, prompting for a username. Your CLI may be used in scripts, so use caution in prompting users that have not specified this switch.

- `-o` for `--output`.

Some commands produce files as the result of their execution. This option should be used to help determine where those files should be located. In cases where a single file is created, this option should be a file path. In cases where many files are created, this option should be a directory path.

- `-v` for `--verbosity`.

Commands often provide output to the user on the console; this option is used to specify the amount of output the user requests. For more information, see [The --verbosity option](#) later in this article.

There are also some aliases with common usage limited to the .NET CLI. You can use these aliases for other options in your apps, but be aware of the possibility of confusion.

- `-c` for `--configuration`

This option often refers to a named Build Configuration, like `Debug` or `Release`. You can use any name you want for a configuration, but most tools are expecting one of those. This setting is often used to configure other properties in a way that makes sense for that configuration—for example, doing less code optimization when building the `Debug` configuration. Consider this option if your command has different modes of operation.

- `-f` for `--framework`

This option is used to select a single [Target Framework Moniker \(TFM\)](#) to execute for, so if your CLI application has differing behavior based on which TFM is chosen, you should support this flag.

- `-p` for `--property`

If your application eventually invokes MSBuild, the user will often need to customize that call in some way. This option allows for MSBuild properties to be

provided on the command line and passed on to the underlying MSBuild call. If your app doesn't use MSBuild but needs a set of key-value pairs, consider using this same option name to take advantage of users' expectations.

- `-r` for `--runtime`

If your application can run on different runtimes, or has runtime-specific logic, consider supporting this option as a way of specifying a [Runtime Identifier](#). If your app supports `--runtime`, consider supporting `--os` and `--arch` also. These options let you specify just the OS or the architecture parts of the RID, leaving the part not specified to be determined from the current platform. For more information, see [dotnet publish](#).

Short names

Make names for commands, options, and arguments as short and easy to spell as possible. For example, if `class` is clear enough don't make the command `classification`.

Lowercase names

Define names in lowercase only, except you can make uppercase aliases to make commands or options case insensitive.

Kebab case names

Use [kebab case](#) to distinguish words. For example, `--additional-probing-path`.

Pluralization

Within an app, be consistent in pluralization. For example, don't mix plural and singular names for options that can have multiple values (maximum arity greater than one):

[+] [Expand table](#)

| Option names | Consistency |
|--|-------------|
| <code>--additional-probing-paths</code> and <code>--sources</code> | ✓ |
| <code>--additional-probing-path</code> and <code>--source</code> | ✓ |
| <code>--additional-probing-paths</code> and <code>--source</code> | ✗ |

| Option names | Consistency |
|---|-------------|
| --additional-probing-path and --sources | ✗ |

Verbs vs. nouns

Use verbs rather than nouns for commands that refer to actions (those without subcommands under them), for example: `dotnet workload remove`, not `dotnet workload removal`. And use nouns rather than verbs for options, for example: `--configuration`, not `--configure`.

The `--verbosity` option

`System.CommandLine` applications typically offer a `--verbosity` option that specifies how much output is sent to the console. Here are the standard five settings:

- `Q[uiet]`
- `M[inimal]`
- `N[ormal]`
- `D[etailed]`
- `Diag[nostic]`

These are the standard names, but existing apps sometimes use `Silent` in place of `Quiet`, and `Trace`, `Debug`, or `Verbose` in place of `Diagnostic`.

Each app defines its own criteria that determine what gets displayed at each level. Typically an app only needs three levels:

- Quiet
- Normal
- Diagnostic

If an app doesn't need five different levels, the option should still define the same five settings. In that case, `Minimal` and `Normal` will produce the same output, and `Detailed` and `Diagnostic` will likewise be the same. This allows your users to just type what they are familiar with, and the best fit will be used.

The expectation for `Quiet` is that no output is displayed on the console. However, if an app offers an interactive mode, the app should do one of the following alternatives:

- Display prompts for input when `--interactive` is specified, even if `--verbosity` is `Quiet`.

- Disallow the use of `--verbosity Quiet` and `--interactive` together.

Otherwise the app will wait for input without telling the user what it's waiting for. It will appear that your application froze and the user will have no idea the application is waiting for input.

If you define aliases, use `-v` for `--verbosity` and make `-v` without an argument an alias for `--verbosity Diagnostic`. Use `-q` for `--verbosity Quiet`.

The .NET CLI and POSIX conventions

The .NET CLI does not consistently follow all POSIX conventions.

Double-dash

Several commands in the .NET CLI have a special implementation of the double-dash token. In the case of `dotnet run`, `dotnet watch`, and `dotnet tool run`, tokens that follow `--` are passed to the app that is being run by the command. For example:

```
.NET CLI  
dotnet run --project ./myapp.csproj -- --message "Hello world!"  
      ^^
```

In this example, the `--project` option is passed to the `dotnet run` command, and the `--message` option with its argument is passed as a command-line option to *myapp* when it runs.

The `--` token is not always required for passing options to an app that you run by using `dotnet run`. Without the double-dash, the `dotnet run` command automatically passes on to the app being run any options that aren't recognized as applying to `dotnet run` itself or to MSBuild. So the following command lines are equivalent because `dotnet run` doesn't recognize the arguments and options:

```
.NET CLI  
dotnet run -- quotes read --delay 0 --fg-color red  
dotnet run quotes read --delay 0 --fg-color red
```

Omission of the option-to-argument delimiter

The .NET CLI doesn't support the POSIX convention that lets you omit the delimiter when you are specifying a single-character option alias.

Multiple arguments without repeating the option name

The .NET CLI doesn't accept multiple arguments for one option without repeating the option name.

Boolean options

In the .NET CLI, some Boolean options result in the same behavior when you pass `false` as when you pass `true`. This behavior results when .NET CLI code that implements the option only checks for the presence or absence of the option, ignoring the value. An example is `--no-restore` for the `dotnet build` command. Pass `no-restore false` and the restore operation will be skipped the same as when you specify `no-restore true` or `no-restore`.

Kebab case

In some cases, the .NET CLI doesn't use kebab case for command, option, or argument names. For example, there is a .NET CLI option that is named `--additionalprobingpath` instead of `--additional-probing-path`.

See also

- [Open-source CLI design guidance](#) ↗
- [GNU standards](#) ↗
- [System.CommandLine overview](#)
- [Tutorial: Get started with System.CommandLine](#)

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For



.NET feedback

.NET is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

more information, see [our contributor guide](#).

How to define commands, options, and arguments in System.CommandLine

Article • 05/25/2023

ⓘ Important

`System.CommandLine` is currently in PREVIEW, and this documentation is for version 2.0 beta 4. Some information relates to prerelease product that may be substantially modified before it's released. Microsoft makes no warranties, express or implied, with respect to the information provided here.

This article explains how to define [commands](#), [options](#), and [arguments](#) in command-line apps that are built with the `System.CommandLine` library. To build a complete application that illustrates these techniques, see the tutorial [Get started with System.CommandLine](#).

For guidance on how to design a command-line app's commands, options, and arguments, see [Design guidance](#).

Define a root command

Every command-line app has a [root command](#), which refers to the executable file itself. The simplest case for invoking your code, if you have an app with no subcommands, options, or arguments, would look like this:

C#

```
using System.CommandLine;

class Program
{
    static async Task Main(string[] args)
    {
        var rootCommand = new RootCommand("Sample command-line app");

        rootCommand.SetHandler(() =>
        {
            Console.WriteLine("Hello world!");
        });

        await rootCommand.InvokeAsync(args);
    }
}
```

Define subcommands

Commands can have child commands, known as *subcommands* or *verbs*, and they can nest as many levels as you need. You can add subcommands as shown in the following example:

C#

```
var rootCommand = new RootCommand();
var sub1Command = new Command("sub1", "First-level subcommand");
rootCommand.Add(sub1Command);
var sub1aCommand = new Command("sub1a", "Second level subcommand");
sub1Command.Add(sub1aCommand);
```

The innermost subcommand in this example can be invoked like this:

Console

```
myapp sub1 sub1a
```

Define options

A command handler method typically has parameters, and the values can come from command-line *options*. The following example creates two options and adds them to the root command. The option names include double-hyphen prefixes, which is [typical for POSIX CLIs](#). The command handler code displays the values of those options:

C#

```
var delayOption = new Option<int>
    (name: "--delay",
     description: "An option whose argument is parsed as an int.",
     getDefaultValue: () => 42);
var messageOption = new Option<string>
    ("--message", "An option whose argument is parsed as a string.");

var rootCommand = new RootCommand();
rootCommand.Add(delayOption);
rootCommand.Add(messageOption);

rootCommand.SetHandler((delayOptionValue, messageOptionValue) =>
{
    Console.WriteLine($"--delay = {delayOptionValue}");
    Console.WriteLine($"--message = {messageOptionValue}");
},
delayOption, messageOption);
```

Here's an example of command-line input and the resulting output for the preceding example code:

```
Console
myapp --delay 21 --message "Hello world!"


Output
--delay = 21
--message = Hello world!
```

Global options

To add an option to one command at a time, use the `Add` or `AddOption` method as shown in the preceding example. To add an option to a command and recursively to all of its subcommands, use the `AddGlobalOption` method, as shown in the following example:

```
C#
var delayOption = new Option<int>
    ("--delay", "An option whose argument is parsed as an int.");
var messageOption = new Option<string>
    ("--message", "An option whose argument is parsed as a string.");

var rootCommand = new RootCommand();
rootCommand.AddGlobalOption(delayOption);
rootCommand.Add(messageOption);

var subCommand1 = new Command("sub1", "First level subcommand");
rootCommand.Add(subCommand1);

var subCommand1a = new Command("sub1a", "Second level subcommand");
subCommand1.Add(subCommand1a);

subCommand1a.SetHandler((delayOptionValue) =>
{
    Console.WriteLine($"--delay = {delayOptionValue}");
},
delayOption);

await rootCommand.InvokeAsync(args);
```

The preceding code adds `--delay` as a global option to the root command, and it's available in the handler for `subCommand1a`.

Define arguments

Arguments are defined and added to commands like options. The following example is like the options example, but it defines arguments instead of options:

C#

```
var delayArgument = new Argument<int>
    (name: "delay",
     description: "An argument that is parsed as an int.",
     getDefaultValue: () => 42);
var messageArgument = new Argument<string>
    ("message", "An argument that is parsed as a string.");

var rootCommand = new RootCommand();
rootCommand.Add(delayArgument);
rootCommand.Add(messageArgument);

rootCommand.SetHandler((delayArgumentValue, messageArgumentValue) =>
{
    Console.WriteLine($"<delay> argument = {delayArgumentValue}");
    Console.WriteLine($"<message> argument = {messageArgumentValue}");
},
delayArgument, messageArgument);

await rootCommand.InvokeAsync(args);
```

Here's an example of command-line input and the resulting output for the preceding example code:

Console

```
myapp 42 "Hello world!"
```

Output

```
<delay> argument = 42
<message> argument = Hello world!
```

An argument that is defined without a default value, such as `messageArgument` in the preceding example, is treated as a required argument. An error message is displayed, and the command handler isn't called, if a required argument isn't provided.

Define aliases

Both commands and options support [aliases](#). You can add an alias to an option by calling `AddAlias`:

```
C#  
  
var option = new Option("--framework");  
option.AddAlias("-f");
```

Given this alias, the following command lines are equivalent:

```
Console  
  
myapp -f net6.0  
myapp --framework net6.0
```

Command aliases work the same way.

```
C#  
  
var command = new Command("serialize");  
command.AddAlias("serialise");
```

This code makes the following command lines equivalent:

```
Console  
  
myapp serialize  
myapp serialise
```

We recommend that you minimize the number of option aliases that you define, and avoid defining certain aliases in particular. For more information, see [Short-form aliases](#).

Required options

To make an option required, set its `IsRequired` property to `true`, as shown in the following example:

```
C#  
  
var endpointOption = new Option<Uri>("--endpoint") { IsRequired = true };  
var command = new RootCommand();  
command.Add(endpointOption);  
  
command.SetHandler((uri) =>  
{
```

```
        Console.WriteLine(uri?.GetType());
        Console.WriteLine(uri?.ToString());
    },
    endpointOption);

await command.InvokeAsync(args);
```

The options section of the command help indicates the option is required:

Output

```
Options:
--endpoint <uri> (REQUIRED)
--version           Show version information
-?, -h, --help      Show help and usage information
```

If the command line for this example app doesn't include `--endpoint`, an error message is displayed and the command handler isn't called:

Output

```
Option '--endpoint' is required.
```

If a required option has a default value, the option doesn't have to be specified on the command line. In that case, the default value provides the required option value.

Hidden commands, options, and arguments

You might want to support a command, option, or argument, but avoid making it easy to discover. For example, it might be a deprecated or administrative or preview feature. Use the `IsHidden` property to prevent users from discovering such features by using tab completion or help, as shown in the following example:

C#

```
var endpointOption = new Option<Uri>("--endpoint") { IsHidden = true };
var command = new RootCommand();
command.Add(endpointOption);

command.SetHandler((uri) =>
{
    Console.WriteLine(uri?.GetType());
    Console.WriteLine(uri?.ToString());
},
endpointOption);
```

```
await command.InvokeAsync(args);
```

The options section of this example's command help omits the `--endpoint` option.

Output

Options:

| | |
|-----------------------------|---------------------------------|
| <code>--version</code> | Show version information |
| <code>-?, -h, --help</code> | Show help and usage information |

Set argument arity

You can explicitly set argument [arity](#) by using the `Arity` property, but in most cases that is not necessary. `System.CommandLine` automatically determines the argument arity based on the argument type:

[+] Expand table

| Argument type | Default arity |
|----------------------|---------------------------------------|
| <code>Boolean</code> | <code>ArgumentArity.ZeroOrOne</code> |
| Collection types | <code>ArgumentArity.ZeroOrMore</code> |
| Everything else | <code>ArgumentArity.ExactlyOne</code> |

Multiple arguments

By default, when you call a command, you can repeat an option name to specify multiple arguments for an option that has maximum [arity](#) greater than one.

Console

```
myapp --items one --items two --items three
```

To allow multiple arguments without repeating the option name, set `Option.AllowMultipleArgumentsPerToken` to `true`. This setting lets you enter the following command line.

Console

```
myapp --items one two three
```

The same setting has a different effect if maximum argument arity is 1. It allows you to repeat an option but takes only the last value on the line. In the following example, the value `three` would be passed to the app.

Console

```
myapp --item one --item two --item three
```

List valid argument values

To specify a list of valid values for an option or argument, specify an enum as the option type or use [FromAmong](#), as shown in the following example:

C#

```
var languageOption = new Option<string>(
    "--language",
    "An option that must be one of the values of a static list.")
    .FromAmong(
        "csharp",
        "fsharp",
        "vb",
        "pwsh",
        "sql");
```

Here's an example of command-line input and the resulting output for the preceding example code:

Console

```
myapp --language not-a-language
```

Output

```
Argument 'not-a-language' not recognized. Must be one of:
  'csharp'
  'fsharp'
  'vb'
  'pwsh'
  'sql'
```

The options section of command help shows the valid values:

| Output | |
|--|--|
| Options: | |
| --language <csharp fsharp vb pwsh sql> | An option that must be one of the values of a static list. |
| --version | Show version information |
| -?, -h, --help | Show help and usage information |

Option and argument validation

For information about argument validation and how to customize it, see the following sections in the [Parameter binding](#) article:

- [Built-in type and arity argument validation](#)
- [Custom validation and binding](#)

See also

- [System.CommandLine overview](#)
- [Parameter binding](#)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

.NET is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

How to bind arguments to handlers in System.CommandLine

Article • 05/11/2023

ⓘ Important

`System.CommandLine` is currently in PREVIEW, and this documentation is for version 2.0 beta 4. Some information relates to prerelease product that may be substantially modified before it's released. Microsoft makes no warranties, express or implied, with respect to the information provided here.

The process of parsing arguments and providing them to command handler code is called *parameter binding*. `System.CommandLine` has the ability to bind many argument types built in. For example, integers, enums, and file system objects such as `FileInfo` and `DirectoryInfo` can be bound. Several `System.CommandLine` types can also be bound.

Built-in argument validation

Arguments have expected types and [arity](#). `System.CommandLine` rejects arguments that don't match these expectations.

For example, a parse error is displayed if the argument for an integer option isn't an integer.

Console

```
myapp --delay not-an-int
```

Output

```
Cannot parse argument 'not-an-int' as System.Int32.
```

An arity error is displayed if multiple arguments are passed to an option that has maximum arity of one:

Console

```
myapp --delay-option 1 --delay-option 2
```

Output

```
Option '--delay' expects a single argument but 2 were provided.
```

This behavior can be overridden by setting [Option.AllowMultipleArgumentsPerToken](#) to `true`. In that case you can repeat an option that has maximum arity of one, but only the last value on the line is accepted. In the following example, the value `three` would be passed to the app.

Console

```
myapp --item one --item two --item three
```

Parameter binding up to 8 options and arguments

The following example shows how to bind options to command handler parameters, by calling [SetHandler](#):

C#

```
var delayOption = new Option<int>
    ("--delay", "An option whose argument is parsed as an int.");
var messageOption = new Option<string>
    ("--message", "An option whose argument is parsed as a string.");

var rootCommand = new RootCommand("Parameter binding example");
rootCommand.Add(delayOption);
rootCommand.Add(messageOption);

rootCommand.SetHandler(
    (delayOptionValue, messageOptionValue) =>
{
    DisplayIntAndString(delayOptionValue, messageOptionValue);
},
delayOption, messageOption);

await rootCommand.InvokeAsync(args);
```

C#

```
public static void DisplayIntAndString(int delayOptionValue, string
messageOptionValue)
{
    Console.WriteLine($"--delay = {delayOptionValue}");
```

```
        Console.WriteLine($"--message = {messageOptionValue}");
    }
```

The lambda parameters are variables that represent the values of options and arguments:

C#

```
(delayOptionValue, messageOptionValue) =>
{
    DisplayIntAndString(delayOptionValue, messageOptionValue);
},
```

The variables that follow the lambda represent the option and argument objects that are the sources of the option and argument values:

C#

```
delayOption, messageOption);
```

The options and arguments must be declared in the same order in the lambda and in the parameters that follow the lambda. If the order is not consistent, one of the following scenarios will result:

- If the out-of-order options or arguments are of different types, a run-time exception is thrown. For example, an `int` might appear where a `string` should be in the list of sources.
- If the out-of-order options or arguments are of the same type, the handler silently gets the wrong values in the parameters provided to it. For example, `string` option `x` might appear where `string` option `y` should be in the list of sources. In that case, the variable for the option `y` value gets the option `x` value.

There are overloads of `SetHandler` that support up to 8 parameters, with both synchronous and asynchronous signatures.

Parameter binding more than 8 options and arguments

To handle more than 8 options, or to construct a custom type from multiple options, you can use `InvocationContext` or a custom binder.

Use `InvocationContext`

A `SetHandler` overload provides access to the `InvocationContext` object, and you can use `InvocationContext` to get any number of option and argument values. For examples, see [Set exit codes](#) and [Handle termination](#).

Use a custom binder

A custom binder lets you combine multiple option or argument values into a complex type and pass that into a single handler parameter. Suppose you have a `Person` type:

```
C#  
  
public class Person  
{  
    public string? FirstName { get; set; }  
    public string? LastName { get; set; }  
}
```

Create a class derived from `BinderBase<T>`, where `T` is the type to construct based on command line input:

```
C#  
  
public class PersonBinder : BinderBase<Person>  
{  
    private readonly Option<string> _firstNameOption;  
    private readonly Option<string> _lastNameOption;  
  
    public PersonBinder(Option<string> firstNameOption, Option<string>  
lastNameOption)  
    {  
        _firstNameOption = firstNameOption;  
        _lastNameOption = lastNameOption;  
    }  
  
    protected override Person GetBoundValue(BindingContext bindingContext)  
=>  
    new Person  
    {  
        FirstName =  
bindingContext.ParseResult.GetValueForOption(_firstNameOption),  
        LastName =  
bindingContext.ParseResult.GetValueForOption(_lastNameOption)  
    };  
}
```

With the custom binder, you can get your custom type passed to your handler the same way you get values for options and arguments:

```
C#
```

```
rootCommand.SetHandler((fileOptionValue, person) =>
{
    DoRootCommand(fileOptionValue, person);
},
fileOption, new PersonBinder(firstNameOption, lastNameOption));
```

Here's the complete program that the preceding examples are taken from:

```
C#
```

```
using System.CommandLine;
using System.CommandLine.Binding;

public class Program
{
    internal static async Task Main(string[] args)
    {
        var fileOption = new Option<FileInfo?>(
            name: "--file",
            description: "An option whose argument is parsed as a
FileInfo",
            getDefaultValue: () => new
FileInfo("scl.runtimeconfig.json"));

        var firstNameOption = new Option<string>(
            name: "--first-name",
            description: "Person.FirstName");

        var lastNameOption = new Option<string>(
            name: "--last-name",
            description: "Person.LastName");

        var rootCommand = new RootCommand();
        rootCommand.Add(fileOption);
        rootCommand.Add(firstNameOption);
        rootCommand.Add(lastNameOption);

        rootCommand.SetHandler((fileOptionValue, person) =>
        {
            DoRootCommand(fileOptionValue, person);
},
fileOption, new PersonBinder(firstNameOption, lastNameOption));

        await rootCommand.InvokeAsync(args);
    }

    public static void DoRootCommand(FileInfo? aFile, Person aPerson)
```

```

    {
        Console.WriteLine($"File = {aFile?.FullName}");
        Console.WriteLine($"Person = {aPerson?.FirstName}
{aPerson?.LastName}");
    }

    public class Person
    {
        public string? FirstName { get; set; }
        public string? LastName { get; set; }
    }

    public class PersonBinder : BinderBase<Person>
    {
        private readonly Option<string> _firstNameOption;
        private readonly Option<string> _lastNameOption;

        public PersonBinder(Option<string> firstNameOption, Option<string>
lastNameOption)
        {
            _firstNameOption = firstNameOption;
            _lastNameOption = lastNameOption;
        }

        protected override Person GetBoundValue(BindingContext
bindingContext) =>
            new Person
            {
                FirstName =
bindingContext.ParseResult.GetValueForOption(_firstNameOption),
                LastName =
bindingContext.ParseResult.GetValueForOption(_lastNameOption)
            };
    }
}

```

Set exit codes

There are [Task](#)-returning [Func](#) overloads of [SetHandler](#). If your handler is called from [async](#) code, you can return a [Task<int>](#) from a handler that uses one of these, and use the `int` value to set the process exit code, as in the following example:

C#

```

static async Task<int> Main(string[] args)
{
    var delayOption = new Option<int>("--delay");
    var messageOption = new Option<string>("--message");

    var rootCommand = new RootCommand("Parameter binding example");
    rootCommand.Add(delayOption);

```

```

rootCommand.Add(messageOption);

rootCommand.SetHandler((delayOptionValue, messageOptionValue) =>
{
    Console.WriteLine($"--delay = {delayOptionValue}");
    Console.WriteLine($"--message = {messageOptionValue}");
    return Task.FromResult(100);
},
delayOption, messageOption);

return await rootCommand.InvokeAsync(args);
}

```

However, if the lambda itself needs to be async, you can't return a `Task<int>`. In that case, use `InvocationContext.ExitCode`. You can get the `InvocationContext` instance injected into your lambda by using a `SetHandler` overload that specifies the `InvocationContext` as the sole parameter. This `SetHandler` overload doesn't let you specify `IValueDescriptor<T>` objects, but you can get option and argument values from the `ParseResult` property of `InvocationContext`, as shown in the following example:

C#

```

static async Task<int> Main(string[] args)
{
    var delayOption = new Option<int>("--delay");
    var messageOption = new Option<string>("--message");

    var rootCommand = new RootCommand("Parameter binding example");
    rootCommand.Add(delayOption);
    rootCommand.Add(messageOption);

    rootCommand.SetHandler(async (context) =>
    {
        int delayOptionValue =
context.ParseResult.GetValueForOption(delayOption);
        string? messageOptionValue =
context.ParseResult.GetValueForOption(messageOption);

        Console.WriteLine($"--delay = {delayOptionValue}");
        await Task.Delay(delayOptionValue);
        Console.WriteLine($"--message = {messageOptionValue}");
        context.ExitCode = 100;
    });

    return await rootCommand.InvokeAsync(args);
}

```

If you don't have asynchronous work to do, you can use the `Action` overloads. In that case, set the exit code by using `InvocationContext.ExitCode` the same way you would

with an async lambda.

The exit code defaults to 1. If you don't set it explicitly, its value is set to 0 when your handler exits normally. If an exception is thrown, it keeps the default value.

Supported types

The following examples show code that binds some commonly used types.

Enums

The values of `enum` types are bound by name, and the binding is case insensitive:

```
C#  
  
var colorOption = new Option<ConsoleColor>("--color");  
  
var rootCommand = new RootCommand("Enum binding example");  
rootCommand.Add(colorOption);  
  
rootCommand.SetHandler((colorOptionValue) =>  
    { Console.WriteLine(colorOptionValue); },  
    colorOption);  
  
await rootCommand.InvokeAsync(args);
```

Here's sample command-line input and resulting output from the preceding example:

Console

```
myapp --color red  
myapp --color RED
```

Output

```
Red  
Red
```

Arrays and lists

Many common types that implement [IEnumerable](#) are supported. For example:

```
C#
```

```
var itemsOption = new Option<IEnumerable<string>>("--items")
{ AllowMultipleArgumentsPerToken = true };

var command = new RootCommand("IEnumarable binding example");
command.Add(itemsOption);

command.SetHandler((items) =>
{
    Console.WriteLine(items.GetType());

    foreach (string item in items)
    {
        Console.WriteLine(item);
    }
},
itemsOption);

await command.InvokeAsync(args);
```

Here's sample command-line input and resulting output from the preceding example:

Console

```
--items one --items two --items three
```

Output

```
System.Collections.Generic.List`1[System.String]
one
two
three
```

Because `AllowMultipleArgumentsPerToken` is set to `true`, the following input results in the same output:

Console

```
--items one two three
```

File system types

Command-line applications that work with the file system can use the [FileSystemInfo](#), [FileInfo](#), and [DirectoryInfo](#) types. The following example shows the use of [FileSystemInfo](#):

C#

```
var fileOrDirectoryOption = new Option<FileSystemInfo>("--file-or-directory");

var command = new RootCommand();
command.Add(fileOrDirectoryOption);

command.SetHandler((fileSystemInfo) =>
{
    switch (fileSystemInfo)
    {
        case FileInfo file : 
            Console.WriteLine($"File name: {file.FullName}");
            break;
        case DirectoryInfo directory:
            Console.WriteLine($"Directory name: {directory.FullName}");
            break;
        default:
            Console.WriteLine("Not a valid file or directory name.");
            break;
    }
},
fileOrDirectoryOption);

await command.InvokeAsync(args);
```

With `FileInfo` and `DirectoryInfo` the pattern matching code is not required:

C#

```
var fileOption = new Option<FileInfo>("--file");

var command = new RootCommand();
command.Add(fileOption);

command.SetHandler((file) =>
{
    if (file is not null)
    {
        Console.WriteLine($"File name: {file?.FullName}");
    }
    else
    {
        Console.WriteLine("Not a valid file name.");
    }
},
fileOption);

await command.InvokeAsync(args);
```

Other supported types

Many types that have a constructor that takes a single string parameter can be bound in this way. For example, code that would work with `FileInfo` works with a `Uri` instead.

C#

```
var endpointOption = new Option<Uri>("--endpoint");

var command = new RootCommand();
command.Add(endpointOption);

command.SetHandler((uri) =>
{
    Console.WriteLine($"URL: {uri?.ToString()}");
},
endpointOption);

await command.InvokeAsync(args);
```

Besides the file system types and `Uri`, the following types are supported:

- `bool`
- `byte`
- `DateTime`
- `DateTimeOffset`
- `decimal`
- `double`
- `float`
- `Guid`
- `int`
- `long`
- `sbyte`
- `short`
- `uint`
- `ulong`
- `ushort`

Use `System.CommandLine` objects

There's a `SetHandler` overload that gives you access to the `InvocationContext` object. That object can then be used to access other `System.CommandLine` objects. For example,

you have access to the following objects:

- [InvocationContext](#)
- [CancellationToken](#)
- [IConsole](#)
- [ParseResult](#)

InvocationContext

For examples, see [Set exit codes](#) and [Handle termination](#).

CancellationToken

For information about how to use [CancellationToken](#), see [How to handle termination](#).

IConsole

[IConsole](#) makes testing as well as many extensibility scenarios easier than using `System.Console`. It's available in the [InvocationContext.Console](#) property.

ParseResult

The [ParseResult](#) object is available in the [InvocationContext.ParseResult](#) property. It's a singleton structure that represents the results of parsing the command line input. You can use it to check for the presence of options or arguments on the command line or to get the [ParseResult.UnmatchedTokens](#) property. This property contains a list of the [tokens](#) that were parsed but didn't match any configured command, option, or argument.

The list of unmatched tokens is useful in commands that behave like wrappers. A wrapper command takes a set of [tokens](#) and forwards them to another command or app. The `sudo` command in Linux is an example. It takes the name of a user to impersonate followed by a command to run. For example:

```
Console
```

```
sudo -u admin apt update
```

This command line would run the `apt update` command as the user `admin`.

To implement a wrapper command like this one, set the command property `TreatUnmatchedTokensAsErrors` to `false`. Then the `ParseResult.UnmatchedTokens` property will contain all of the arguments that don't explicitly belong to the command. In the preceding example, `ParseResult.UnmatchedTokens` would contain the `apt` and `update` tokens. Your command handler could then forward the `UnmatchedTokens` to a new shell invocation, for example.

Custom validation and binding

To provide custom validation code, call `AddValidator` on your command, option, or argument, as shown in the following example:

C#

```
var delayOption = new Option<int>("--delay");
delayOption.AddValidator(result =>
{
    if (result.GetValueForOption(delayOption) < 1)
    {
        result.ErrorMessage = "Must be greater than 0";
    }
});
```

If you want to parse as well as validate the input, use a `ParseArgument<T>` delegate, as shown in the following example:

C#

```
var delayOption = new Option<int>(
    name: "--delay",
    description: "An option whose argument is parsed as an int.",
    isDefault: true,
    parseArgument: result =>
{
    if (!result.Tokens.Any())
    {
        return 42;
    }

    if (int.TryParse(result.Tokens.Single().Value, out var delay))
    {
        if (delay < 1)
        {
            result.ErrorMessage = "Must be greater than 0";
        }
        return delay;
    }
    else
```

```

    {
        result.ErrorMessage = "Not an int.";
        return 0; // Ignored.
    }
});

```

The preceding code sets `isDefault` to `true` so that the `parseArgument` delegate will be called even if the user didn't enter a value for this option.

Here are some examples of what you can do with `ParseArgument<T>` that you can't do with `AddValidator`:

- Parsing of custom types, such as the `Person` class in the following example:

C#

```

public class Person
{
    public string? FirstName { get; set; }
    public string? LastName { get; set; }
}

```

C#

```

var personOption = new Option<Person?>(
    name: "--person",
    description: "An option whose argument is parsed as a Person",
    parseArgument: result =>
{
    if (result.Tokens.Count != 2)
    {
        result.ErrorMessage = "--person requires two arguments";
        return null;
    }
    return new Person
    {
        FirstName = result.Tokens.First().Value,
        LastName = result.Tokens.Last().Value
    };
})
{
    Arity = ArgumentArity.OneOrMore,
    AllowMultipleArgumentsPerToken = true
};

```

- Parsing of other kinds of input strings (for example, parse "1,2,3" into `int[]`).
- Dynamic arity. For example, you have two arguments that are defined as string arrays, and you have to handle a sequence of strings in the command line input.

The [ArgumentResult.OnlyTake](#) method enables you to dynamically divide up the input strings between the arguments.

See also

[System.CommandLine overview](#)

 [Collaborate with us on GitHub](#)

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

.NET is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

Tab completion for System.CommandLine

Article • 03/11/2022

ⓘ Important

`System.CommandLine` is currently in PREVIEW, and this documentation is for version 2.0 beta 4. Some information relates to prerelease product that may be substantially modified before it's released. Microsoft makes no warranties, express or implied, with respect to the information provided here.

Apps that use `System.CommandLine` have built-in support for tab completion in certain shells. To enable it, the end user has to take a few steps once per shell. Once the user does this, tab completion is automatic for static values in your app, such as enum values or values you define by calling [FromAmong](#). You can also customize the tab completion by getting values dynamically at runtime.

Enable tab completion

On the machine where you'd like to enable tab completion, do the following steps.

For the .NET CLI:

- See [How to enable tab completion](#).

For other command-line apps built on `System.CommandLine`:

- Install the [dotnet-suggest](#) global tool.
- Add the appropriate shim script to your shell profile. You may have to create a shell profile file. The shim script forwards completion requests from your shell to the `dotnet-suggest` tool, which delegates to the appropriate `System.CommandLine`-based app.
 - For `bash`, add the contents of [dotnet-suggest-shim.bash](#) to `~/.bash_profile`.
 - For `zsh`, add the contents of [dotnet-suggest-shim.zsh](#) to `~/.zshrc`.
 - For PowerShell, add the contents of [dotnet-suggest-shim.ps1](#) to your PowerShell profile. You can find the expected path to your PowerShell profile by

running the following command in your console:

```
Console
```

```
echo $profile
```

Once the user's shell is set up, completions will work for all apps that are built by using `System.CommandLine`.

For `cmd.exe` on Windows (the Windows Command Prompt) there is no pluggable tab completion mechanism, so no shim script is available. For other shells, [look for a GitHub issue that is labeled Area-Completions ↗](#). If you don't find an issue, you can [open a new one ↗](#).

Get tab completion values at run-time

The following code shows an app that gets values for tab completion dynamically at runtime. The code gets a list of the next two weeks of dates following the current date. The list is provided to the `--date` option by calling `AddCompletions`:

```
C#
```

```
using System.CommandLine;
using System.CommandLine.Completions;
using System.CommandLine.Parsing;

await new DateCommand().InvokeAsync(args);

class DateCommand : Command
{
    private Argument<string> subjectArgument =
        new ("subject", "The subject of the appointment.");
    private Option<DateTime> dateOption =
        new ("--date", "The day of week to schedule. Should be within one
week.");

    public DateCommand() : base("schedule", "Makes an appointment for
sometime in the next week.")
    {
        this.AddArgument(subjectArgument);
        this.AddOption(dateOption);

        dateOption.AddCompletions((ctx) => {
            var today = System.DateTime.Today;
            var dates = new List<CompletionItem>();
            foreach (var i in Enumerable.Range(1, 7))
            {
                var date = today.AddDays(i);
                dates.Add(new CompletionItem(date.ToString("ddd")));
            }
            return dates;
        });
    }
}
```

```

        dates.Add(new CompletionItem(
            label: date.ToShortDateString(),
            sortText: ${i:2}"));
    }
    return dates;
});

this.SetHandler((subject, date) =>
{
    Console.WriteLine($"Scheduled \'{subject}\' for {date}");
},
subjectArgument, dateOption);
}
}

```

The values shown when the tab key is pressed are provided as `CompletionItem` instances:

C#

```

dates.Add(new CompletionItem(
    label: date.ToShortDateString(),
    sortText: ${i:2}"));

```

The following `CompletionItem` properties are set:

- `Label` is the completion value to be shown.
- `SortText` ensures that the values in the list are presented in the right order. It's set by converting `i` to a two-digit string, so that sorting is based on 01, 02, 03, and so on, through 14. If you don't set this parameter, sorting is based on the `Label`, which in this example is in short date format and won't sort correctly.

There are other `CompletionItem` properties, such as `Documentation` and `Detail`, but they aren't used yet in `System.CommandLine`.

The dynamic tab completion list created by this code also appears in help output:

Output

Description:
Makes an appointment for sometime in the next week.

Usage:
`schedule <subject> [options]`

Arguments:
`<subject>` The subject of the appointment.

```
Options:  
--date  
The day of week to schedule. Should be within one week.  
<2/4/2022|2/5/2022|2/6/2022|2/7/2022|2/8/2022|2/9/2022|2/10/2022>  
--version  
Show version information  
-?, -h, --help
```

See also

[System.CommandLine overview](#)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

How to configure dependency injection in System.CommandLine

Article • 05/11/2023

ⓘ Important

`System.CommandLine` is currently in PREVIEW, and this documentation is for version 2.0 beta 4. Some information relates to prerelease product that may be substantially modified before it's released. Microsoft makes no warranties, express or implied, with respect to the information provided here.

Use a [custom binder](#) to inject custom types into a command handler.

We recommend handler-specific dependency injection (DI) for the following reasons:

- Command-line apps are often short-lived processes, in which startup cost can have a noticeable impact on performance. Optimizing performance is particularly important when tab completions have to be calculated. Command-line apps are unlike Web and GUI apps, which tend to be relatively long-lived processes. Unnecessary startup time is not appropriate for short-lived processes.
- When a command-line app that has multiple subcommands is run, only one of those subcommands will be executed. If an app configures dependencies for the subcommands that don't run, it needlessly degrades performance.

To configure DI, create a class that derives from `BinderBase<T>` where `T` is the interface that you want to inject an instance for. In the `GetBoundValue` method override, get and return the instance you want to inject. The following example injects the default logger implementation for `ILogger`:

C#

```
public class MyCustomBinder : BinderBase<ILogger>
{
    protected override ILogger GetBoundValue(
        BindingContext bindingContext) => GetLogger(bindingContext);

    ILogger GetLogger(BindingContext bindingContext)
    {
        using ILoggerFactory loggerFactory = LoggerFactory.Create(
            builder => builder.AddConsole());
        ILogger logger = loggerFactory.CreateLogger("LoggerCategory");
        return logger;
    }
}
```

```
    }  
}
```

When calling the `SetHandler` method, pass to the lambda an instance of the injected class and pass an instance of your binder class in the list of services:

C#

```
rootCommand.SetHandler(async (fileOptionValue, logger) =>  
{  
    await DoRootCommand(fileOptionValue!, logger);  
},  
fileOption, new MyCustomBinder());
```

The following code is a complete program that contains the preceding examples:

C#

```
using System.CommandLine;  
using System.CommandLine.Binding;  
using Microsoft.Extensions.Logging;  
  
class Program  
{  
    static async Task Main(string[] args)  
    {  
        var fileOption = new Option<FileInfo?>(  
            name: "--file",  
            description: "An option whose argument is parsed as a  
FileInfo");  
  
        var rootCommand = new RootCommand("Dependency Injection sample");  
        rootCommand.Add(fileOption);  
  
        rootCommand.SetHandler(async (fileOptionValue, logger) =>  
        {  
            await DoRootCommand(fileOptionValue!, logger);  
        },  
        fileOption, new MyCustomBinder());  
  
        await rootCommand.InvokeAsync("--file scl.runtimeconfig.json");  
    }  
  
    public static async Task DoRootCommand(FileInfo aFile, ILogger logger)  
    {  
        Console.WriteLine($"File = {aFile?.FullName}");  
        logger.LogCritical("Test message");  
        await Task.Delay(1000);  
    }  
  
    public class MyCustomBinder : BinderBase<ILogger>
```

```
{  
    protected override ILogger GetBoundValue(  
        BindingContext bindingContext) => GetLogger(bindingContext);  
  
    ILogger GetLogger(BindingContext bindingContext)  
    {  
        using ILoggerFactory loggerFactory = LoggerFactory.Create(  
            builder => builder.AddConsole());  
        ILogger logger = loggerFactory.CreateLogger("LoggerCategory");  
        return logger;  
    }  
}
```

See also

[System.CommandLine overview](#)

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

.NET is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

How to customize help in apps that are built with the System.Commandline library

Article • 07/21/2022

You can customize help for a specific command, option, or argument, and you can add or replace whole help sections.

The examples in this article work with the following command-line application:

This code requires a `using` directive:

C#

```
using System.CommandLine;
```

C#

```
var fileOption = new Option<FileInfo>(
    "--file",
    description: "The file to print out.",
    getDefaultValue: () => new FileInfo("scl.runtimeconfig.json"));
var lightModeOption = new Option<bool> (
    "--light-mode",
    description: "Determines whether the background color will be black or white");
var foregroundColorOption = new Option<ConsoleColor>(
    "--color",
    description: "Specifies the foreground color of console output",
    getDefaultValue: () => ConsoleColor.White);

var rootCommand = new RootCommand("Read a file")
{
    fileOption,
    lightModeOption,
    foregroundColorOption
};

rootCommand.SetHandler((file, lightMode, color) =>
{
    Console.BackgroundColor = lightMode ? ConsoleColor.White:
    ConsoleColor.Black;
    Console.ForegroundColor = color;
    Console.WriteLine($"--file = {file?.FullName}");
    Console.WriteLine($"File
contents:\n{file?.OpenText().ReadToEnd()}");
},
```

```
    fileOption,  
    lightModeOption,  
    foregroundColorOption);  
  
    await rootCommand.InvokeAsync(args);
```

Without customization, the following help output is produced:

Output

```
Description:  
  Read a file  
  
Usage:  
  scl [options]  
  
Options:  
  --file <file>                                     The file to  
  print out. [default: scl.runtimeconfig.json]  
  --light-mode                                         Determines  
  whether the background color will be black or  
  white  
  --color                                              Specifies the  
  foreground color of console output  
  <Black|Blue|Cyan|DarkBlue|DarkCyan|DarkGray|Dark  [default:  
  White]  
  Magenta|DarkRed|DarkYellow|Gray|Green|Magenta|Red|White|Ye  
  llow>  
  --version                                            Show version  
  information  
  -?, -h, --help                                         Show help and  
  usage information
```

Customize help for a single option or argument

ⓘ Important

`System.CommandLine` is currently in PREVIEW, and this documentation is for version 2.0 beta 4. Some information relates to prerelease product that may be substantially modified before it's released. Microsoft makes no warranties, express or implied, with respect to the information provided here.

To customize the name of an option's argument, use the option's [ArgumentHelpName](#) property. And [HelpBuilder.CustomizeSymbol](#) lets you customize several parts of the help output for a command, option, or argument ([Symbol](#) is the base class for all three types). With [CustomizeSymbol](#), you can specify:

- The first column text.
- The second column text.
- The way a default value is described.

In the sample app, `--light-mode` is explained adequately, but changes to the `--file` and `--color` option descriptions will be helpful. For `--file`, the argument can be identified as a `<FILEPATH>` instead of `<file>`. For the `--color` option, you can shorten the list of available colors in column one, and in column two you can add a warning that some colors won't work with some backgrounds.

To make these changes, delete the `await rootCommand.InvokeAsync(args);` line shown in the preceding code and add in its place the following code:

C#

```
fileOption.ArgumentHelpName = "FILEPATH";

var parser = new CommandLineBuilder(rootCommand)
    .UseDefaults()
    .UseHelp(ctx =>
{
    ctx.HelpBuilder.CustomizeSymbol(foregroundColorOption,
        firstColumnText: "--color <Black, White, Red, or Yellow>",
        secondColumnText: "Specifies the foreground color. " +
            "Choose a color that provides enough contrast " +
            "with the background color. " +
            "For example, a yellow foreground can't be read " +
            "against a light mode background.");
})
.Build();

parser.Invoke(args);
```

The updated code requires additional `using` directives:

C#

```
using System.CommandLine.Builder;
using System.CommandLine.Help;
using System.CommandLine.Parsing;
```

The app now produces the following help output:

Output

```
Description:
  Read a file
```

```

Usage:
  scl [options]

Options:
  --file <FILEPATH>                                The file to print out. [default:
  CustomHelp.runtimeconfig.json]
  --light-mode                                         Determines whether the background
  color will be black or white
  --color <Black, White, Red, or Yellow>   Specifies the foreground color.
  Choose a color that provides enough contrast
                                              with the background color. For
  example, a yellow foreground can't be read
                                              against a light mode background.
  --version                                            Show version information
  -?, -h, --help                                         Show help and usage information

```

This output shows that the `firstColumnText` and `secondColumnText` parameters support word wrapping within their columns.

Add or replace help sections

You can add or replace a whole section of the help output. For example, suppose you want to add some ASCII art to the description section by using the [Spectre.Console](#) NuGet package.

Change the layout by adding a call to `HelpBuilder.CustomizeLayout` in the lambda passed to the `UseHelp` method:

```

C#

fileOption.ArgumentHelpName = "FILEPATH";

var parser = new CommandLineBuilder(rootCommand)
    .UseDefaults()
    .UseHelp(ctx =>
{
    ctx.HelpBuilder.CustomizeSymbol(foregroundColorOption,
        firstColumnText: "--color <Black, White, Red, or Yellow>",
        secondColumnText: "Specifies the foreground color. " +
            "Choose a color that provides enough contrast " +
            "with the background color. " +
            "For example, a yellow foreground can't be read " +
            "against a light mode background.");
}

```

```

    ctx.HelpBuilder.CustomizeLayout(
        _ =>
            HelpBuilder.Default
                .GetLayout()
                .Skip(1) // Skip the default command description
section.
                .Prepend(
                    _ => Spectre.Console.AnsiConsole.Write(
                        new FigletText(rootCommand.Description!)))
            ));
    }
    .Build();

await parser.InvokeAsync(args);

```

The preceding code requires an additional `using` directive:

C#

```
using Spectre.Console;
```

The [System.CommandLine.Help.HelpBuilder.Default](#) class lets you reuse pieces of existing help formatting functionality and compose them into your custom help.

The help output now looks like this:

Output



Usage:

```
scl [options]
```

Options:

```
--file <FILEPATH>
```

The file to print out. [default:

```
CustomHelp.runtimeconfig.json]
```

```
--light-mode
```

Determines whether the background

color will be black or white

```
--color <Black, White, Red, or Yellow>
```

Specifies the foreground color.

Choose a color that provides enough contrast

with the background color. For

example, a yellow foreground can't be read

against a light mode background.

```
--version
```

Show version information

```
-?, -h, --help
```

Show help and usage information

If you want to just use a string as the replacement section text instead of formatting it with `Spectre.Console`, replace the `Prepend` code in the preceding example with the following code:

C#

```
.Prepend(  
    _ => _.Output.WriteLine(""))
```

See also

[System.CommandLine overview](#)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

How to handle termination in System.CommandLine

Article • 06/27/2023

ⓘ Important

`System.CommandLine` is currently in PREVIEW, and this documentation is for version 2.0 beta 4. Some information relates to prerelease product that may be substantially modified before it's released. Microsoft makes no warranties, express or implied, with respect to the information provided here.

To handle termination, inject a [CancellationToken](#) instance into your handler code. This token can then be passed along to async APIs that you call from within your handler, as shown in the following example:

C#

```
static async Task<int> Main(string[] args)
{
    int returnCode = 0;

    var urlOption = new Option<string>("--url", "A URL.");

    var rootCommand = new RootCommand("Handle termination example");
    rootCommand.Add(urlOption);

    rootCommand.SetHandler(async (context) =>
    {
        string? urlOptionValue =
        context.ParseResult.GetValueForOption(urlOption);
        var token = context.GetCancellationToken();
        returnCode = await DoRootCommand(urlOptionValue, token);
    });

    await rootCommand.InvokeAsync(args);

    return returnCode;
}

public static async Task<int> DoRootCommand(
    string? urlOptionValue, CancellationToken cancellationToken)
{
    try
    {
        using (var httpClient = new HttpClient())
        {
```

```
        await httpClient.GetAsync(urlOptionValue, cancellationToken);
    }
    return 0;
}
catch (OperationCanceledException)
{
    Console.Error.WriteLine("The operation was aborted");
    return 1;
}
}
```

The preceding code uses a `SetHandler` overload that gets an `InvocationContext` instance rather than one or more `IValueDescriptor<T>` objects. The `InvocationContext` is used to get the `CancellationToken` and `ParseResult` objects. `ParseResult` can provide argument or option values.

To test the sample code, run the command with a URL that will take a moment to load, and before it finishes loading, press `Ctrl + C`. On macOS press `Command + Period(.)`. For example:

.NET CLI

```
testapp --url https://learn.microsoft.com/aspnet/core/fundamentals/minimal-apis
```

Output

```
The operation was aborted
```

Cancellation actions can also be added directly using the `CancellationToken.Register` method.

For information about an alternative way to set the process exit code, see [Set exit codes](#).

See also

[System.CommandLine overview](#)

 Collaborate with us on
GitHub

The source for this content can
be found on GitHub, where you



.NET feedback

The .NET documentation is open
source. Provide feedback [here](#).

can also create and review issues and pull requests. For more information, see [our contributor guide](#).

 [Open a documentation issue](#)

 [Provide product feedback](#)

How to use middleware in System.CommandLine

Article • 04/09/2022

ⓘ Important

`System.CommandLine` is currently in PREVIEW, and this documentation is for version 2.0 beta 4. Some information relates to prerelease product that may be substantially modified before it's released. Microsoft makes no warranties, express or implied, with respect to the information provided here.

This article explains how to work with middleware in command-line apps that are built with the `System.CommandLine` library. Use of middleware is an advanced topic that most `System.CommandLine` users won't need to consider.

Introduction to middleware

While each command has a handler that `System.CommandLine` will route to based on input, there's also a mechanism for short-circuiting or altering the input before your application logic is invoked. In between parsing and invocation, there's a chain of responsibility, which you can customize. A number of built-in features of `System.CommandLine` make use of this capability. This is how the `--help` and `--version` options short-circuit calls to your handler.

Each call in the pipeline can take action based on the `ParseResult` and return early, or choose to call the next item in the pipeline. The `ParseResult` can even be replaced during this phase. The last call in the chain is the handler for the specified command.

Add to the middleware pipeline

You can add a call to this pipeline by calling `CommandLineBuilderExtensions.AddMiddleware`. Here's an example of code that enables a custom `directive`. After creating a root command named `rootCommand`, the code as usual adds options, arguments, and handlers. Then the middleware is added:

C#

```

var commandLineBuilder = new CommandLineBuilder(rootCommand);

commandLineBuilder.AddMiddleware(async (context, next) =>
{
    if (context.ParseResult.Directives.Contains("just-say-hi"))
    {
        context.Console.WriteLine("Hi!");
    }
    else
    {
        await next(context);
    }
});

commandLineBuilder.UseDefaults();
var parser = commandLineBuilder.Build();
await parser.InvokeAsync(args);

```

In the preceding code, the middleware writes out "Hi!" if the directive `[just-say-hi]` is found in the parse result. When this happens, the command's normal handler isn't invoked. It isn't invoked because the middleware doesn't call the `next` delegate.

In the example, `context` is `InvocationContext`, a singleton structure that acts as the "root" of the entire command-handling process. This is the most powerful structure in `System.CommandLine`, in terms of capabilities. There are two main uses for it in middleware:

- It provides access to the `BindingContext`, `Parser`, `Console`, and `HelpBuilder` to retrieve dependencies that a middleware requires for its custom logic.
- You can set the `InvocationResult` or `ExitCode` properties in order to terminate command processing in a short-circuiting manner. An example is the `--help` option, which is implemented in this manner.

Here's the complete program, including required `using` directives.

C#

```

using System.CommandLine;
using System.CommandLine.Builder;
using System.CommandLine.Parsing;

class Program
{
    static async Task Main(string[] args)
    {
        var delayOption = new Option<int>("--delay");
        var messageOption = new Option<string>("--message");

```

```

    var rootCommand = new RootCommand("Middleware example");
    rootCommand.Add(delayOption);
    rootCommand.Add(messageOption);

    rootCommand.SetHandler((delayOptionValue, messageOptionValue) =>
    {
        DoRootCommand(delayOptionValue, messageOptionValue);
    },
    delayOption, messageOption);

    var commandLineBuilder = new CommandLineBuilder(rootCommand);

    commandLineBuilder.AddMiddleware(async (context, next) =>
    {
        if (context.ParseResult.Directives.Contains("just-say-hi"))
        {
            context.Console.WriteLine("Hi!");
        }
        else
        {
            await next(context);
        }
    });
}

commandLineBuilder.UseDefaults();
var parser = commandLineBuilder.Build();
await parser.InvokeAsync(args);
}

public static void DoRootCommand(int delay, string message)
{
    Console.WriteLine($"--delay = {delay}");
    Console.WriteLine($"--message = {message}");
}
}

```

Here's an example command line and resulting output from the preceding code:

Console

```
myapp [just-say-hi] --delay 42 --message "Hello world!"
```

Output

```
Hi!
```

See also

[System.CommandLine overview](#)

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

File and Stream I/O

Article • 09/15/2021

File and stream I/O (input/output) refers to the transfer of data either to or from a storage medium. In .NET, the `System.IO` namespaces contain types that enable reading and writing, both synchronously and asynchronously, on data streams and files. These namespaces also contain types that perform compression and decompression on files, and types that enable communication through pipes and serial ports.

A file is an ordered and named collection of bytes that has persistent storage. When you work with files, you work with directory paths, disk storage, and file and directory names. In contrast, a stream is a sequence of bytes that you can use to read from and write to a backing store, which can be one of several storage mediums (for example, disks or memory). Just as there are several backing stores other than disks, there are several kinds of streams other than file streams, such as network, memory, and pipe streams.

Files and directories

You can use the types in the `System.IO` namespace to interact with files and directories. For example, you can get and set properties for files and directories, and retrieve collections of files and directories based on search criteria.

For path naming conventions and the ways to express a file path for Windows systems, including with the DOS device syntax supported in .NET Core 1.1 and later and .NET Framework 4.6.2 and later, see [File path formats on Windows systems](#).

Here are some commonly used file and directory classes:

- [File](#) - provides static methods for creating, copying, deleting, moving, and opening files, and helps create a [FileStream](#) object.
- [FileInfo](#) - provides instance methods for creating, copying, deleting, moving, and opening files, and helps create a [FileStream](#) object.
- [Directory](#) - provides static methods for creating, moving, and enumerating through directories and subdirectories.
- [DirectoryInfo](#) - provides instance methods for creating, moving, and enumerating through directories and subdirectories.
- [Path](#) - provides methods and properties for processing directory strings in a cross-platform manner.

You should always provide robust exception handling when calling filesystem methods. For more information, see [Handling I/O errors](#).

In addition to using these classes, Visual Basic users can use the methods and properties provided by the [Microsoft.VisualBasic.FileIO.FileSystem](#) class for file I/O.

See [How to: Copy Directories](#), [How to: Create a Directory Listing](#), and [How to: Enumerate Directories and Files](#).

Streams

The abstract base class [Stream](#) supports reading and writing bytes. All classes that represent streams inherit from the [Stream](#) class. The [Stream](#) class and its derived classes provide a common view of data sources and repositories, and isolate the programmer from the specific details of the operating system and underlying devices.

Streams involve three fundamental operations:

- Reading - transferring data from a stream into a data structure, such as an array of bytes.
- Writing - transferring data to a stream from a data source.
- Seeking - querying and modifying the current position within a stream.

Depending on the underlying data source or repository, a stream might support only some of these capabilities. For example, the [PipeStream](#) class does not support seeking. The [CanRead](#), [CanWrite](#), and [CanSeek](#) properties of a stream specify the operations that the stream supports.

Here are some commonly used stream classes:

- [FileStream](#) – for reading and writing to a file.
- [IsolatedStorageFileStream](#) – for reading and writing to a file in isolated storage.
- [MemoryStream](#) – for reading and writing to memory as the backing store.
- [BufferedStream](#) – for improving performance of read and write operations.
- [NetworkStream](#) – for reading and writing over network sockets.
- [PipeStream](#) – for reading and writing over anonymous and named pipes.
- [CryptoStream](#) – for linking data streams to cryptographic transformations.

For an example of working with streams asynchronously, see [Asynchronous File I/O](#).

Readers and writers

The [System.IO](#) namespace also provides types for reading encoded characters from streams and writing them to streams. Typically, streams are designed for byte input and output. The reader and writer types handle the conversion of the encoded characters to and from bytes so the stream can complete the operation. Each reader and writer class is associated with a stream, which can be retrieved through the class's `BaseStream` property.

Here are some commonly used reader and writer classes:

- [BinaryReader](#) and [BinaryWriter](#) – for reading and writing primitive data types as binary values.
- [StreamReader](#) and [StreamWriter](#) – for reading and writing characters by using an encoding value to convert the characters to and from bytes.
- [StringReader](#) and [StringWriter](#) – for reading and writing characters to and from strings.
- [TextReader](#) and [TextWriter](#) – serve as the abstract base classes for other readers and writers that read and write characters and strings, but not binary data.

See [How to: Read Text from a File](#), [How to: Write Text to a File](#), [How to: Read Characters from a String](#), and [How to: Write Characters to a String](#).

Asynchronous I/O operations

Reading or writing a large amount of data can be resource-intensive. You should perform these tasks asynchronously if your application needs to remain responsive to the user. With synchronous I/O operations, the UI thread is blocked until the resource-intensive operation has completed. Use asynchronous I/O operations when developing Windows 8.x Store apps to prevent creating the impression that your app has stopped working.

The asynchronous members contain `Async` in their names, such as the [CopyToAsync](#), [FlushAsync](#), [ReadAsync](#), and [WriteAsync](#) methods. You use these methods with the `async` and `await` keywords.

For more information, see [Asynchronous File I/O](#).

Compression

Compression refers to the process of reducing the size of a file for storage.

Decompression is the process of extracting the contents of a compressed file so they are in a usable format. The [System.IO.Compression](#) namespace contains types for compressing and decompressing files and streams.

The following classes are frequently used when compressing and decompressing files and streams:

- [ZipArchive](#) – for creating and retrieving entries in the zip archive.
- [ZipArchiveEntry](#) – for representing a compressed file.
- [ZipFile](#) – for creating, extracting, and opening a compressed package.
- [ZipFileExtensions](#) – for creating and extracting entries in a compressed package.
- [DeflateStream](#) – for compressing and decompressing streams using the Deflate algorithm.
- [GZipStream](#) – for compressing and decompressing streams in gzip data format.

See [How to: Compress and Extract Files](#).

Isolated storage

Isolated storage is a data storage mechanism that provides isolation and safety by defining standardized ways of associating code with saved data. The storage provides a virtual file system that is isolated by user, assembly, and (optionally) domain. Isolated storage is particularly useful when your application does not have permission to access user files. You can save settings or files for your application in a manner that is controlled by the computer's security policy.

Isolated storage is not available for Windows 8.x Store apps; instead, use application data classes in the [Windows.Storage](#) namespace. For more information, see [Application data](#).

The following classes are frequently used when implementing isolated storage:

- [IsolatedStorage](#) – provides the base class for isolated storage implementations.
- [IsolatedStorageFile](#) – provides an isolated storage area that contains files and directories.

- [IsolatedStorageFileStream](#) - exposes a file within isolated storage.

See [Isolated Storage](#).

I/O operations in Windows Store apps

.NET for Windows 8.x Store apps contains many of the types for reading from and writing to streams; however, this set does not include all the .NET I/O types.

Some important differences to note when using I/O operations in Windows 8.x Store apps:

- Types specifically related to file operations, such as [File](#), [FileInfo](#), [Directory](#) and [DirectoryInfo](#), are not included in the .NET for Windows 8.x Store apps. Instead, use the types in the [Windows.Storage](#) namespace of the Windows Runtime, such as [StorageFile](#) and [StorageFolder](#).
- Isolated storage is not available; instead, use [application data](#).
- Use asynchronous methods, such as [ReadAsync](#) and [WriteAsync](#), to prevent blocking the UI thread.
- The path-based compression types [ZipFile](#) and [ZipFileExtensions](#) are not available. Instead, use the types in the [Windows.Storage.Compression](#) namespace.

You can convert between .NET Framework streams and Windows Runtime streams, if necessary. For more information, see [How to: Convert Between .NET Framework Streams and Windows Runtime Streams](#) or [WindowsRuntimeStreamExtensions](#).

For more information about I/O operations in a Windows 8.x Store app, see [Quickstart: Reading and writing files](#).

I/O and security

When you use the classes in the [System.IO](#) namespace, you must follow operating system security requirements such as access control lists (ACLs) to control access to files and directories. This requirement is in addition to any [FileIOPermission](#) requirements.

You can manage ACLs programmatically. For more information, see [How to: Add or Remove Access Control List Entries](#).

Default security policies prevent Internet or intranet applications from accessing files on the user's computer. Therefore, do not use the I/O classes that require a path to a

physical file when writing code that will be downloaded over the internet or intranet. Instead, use [isolated storage](#) for .NET applications.

A security check is performed only when the stream is constructed. Therefore, do not open a stream and then pass it to less-trusted code or application domains.

Related topics

- [Common I/O Tasks](#)

Provides a list of I/O tasks associated with files, directories, and streams, and links to relevant content and examples for each task.

- [Asynchronous File I/O](#)

Describes the performance advantages and basic operation of asynchronous I/O.

- [Isolated Storage](#)

Describes a data storage mechanism that provides isolation and safety by defining standardized ways of associating code with saved data.

- [Pipes](#)

Describes anonymous and named pipe operations in .NET.

- [Memory-Mapped Files](#)

Describes memory-mapped files, which contain the contents of files on disk in virtual memory. You can use memory-mapped files to edit very large files and to create shared memory for interprocess communication.

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

File path formats on Windows systems

Article • 12/14/2022

Members of many of the types in the [System.IO](#) namespace include a `path` parameter that lets you specify an absolute or relative path to a file system resource. This path is then passed to [Windows file system APIs](#). This topic discusses the formats for file paths that you can use on Windows systems.

Traditional DOS paths

A standard DOS path can consist of three components:

- A volume or drive letter followed by the volume separator (`:`).
- A directory name. The [directory separator character](#) separates subdirectories within the nested directory hierarchy.
- An optional filename. The [directory separator character](#) separates the file path and the filename.

If all three components are present, the path is absolute. If no volume or drive letter is specified and the directory name begins with the [directory separator character](#), the path is relative from the root of the current drive. Otherwise, the path is relative to the current directory. The following table shows some possible directory and file paths.

| Path | Description |
|---|---|
| <code>C:\Documents\Newsletters\Summer2018.pdf</code> | An absolute file path from the root of drive <code>C:</code> . |
| <code>\Program Files\Custom Utilities\StringFinder.exe</code> | A relative path from the root of the current drive. |
| <code>2018\January.xlsx</code> | A relative path to a file in a subdirectory of the current directory. |
| <code>..\Publications\TravelBrochure.pdf</code> | A relative path to a file in a directory starting from the current directory. |
| <code>C:\Projects\apilibrary\apilibrary.sln</code> | An absolute path to a file from the root of drive <code>C:</code> . |
| <code>C:Projects\apilibrary\apilibrary.sln</code> | A relative path from the current directory of the <code>C:</code> drive. |

Important

Note the difference between the last two paths. Both specify the optional volume specifier (`C:` in both cases), but the first begins with the root of the specified volume, whereas the second does not. As result, the first is an absolute path from the root directory of drive `C:`, whereas the second is a relative path from the current directory of drive `C:`. Use of the second form when the first is intended is a common source of bugs that involve Windows file paths.

You can determine whether a file path is fully qualified (that is, if the path is independent of the current directory and does not change when the current directory changes) by calling the [Path.IsPathFullyQualified](#) method. Note that such a path can include relative directory segments (`.` and `..`) and still be fully qualified if the resolved path always points to the same location.

The following example illustrates the difference between absolute and relative paths. It assumes that the directory `D:\FY2018\` exists, and that you haven't set any current directory for `D:\` from the command prompt before running the example.

C#

```
using System;
using System.Diagnostics;
using System.IO;
using System.Reflection;

public class Example
{
    public static void Main(string[] args)
    {
        Console.WriteLine($"Current directory is
'{Environment.CurrentDirectory}'");
        Console.WriteLine("Setting current directory to 'C:\\'");

        Directory.SetCurrentDirectory(@"C:\\");
        string path = Path.GetFullPath(@"D:\\FY2018");
        Console.WriteLine($"'D:\\FY2018' resolves to {path}");
        path = Path.GetFullPath(@"D:FY2018");
        Console.WriteLine($"'D:FY2018' resolves to {path}");

        Console.WriteLine("Setting current directory to 'D:\\\\Docs'");
        Directory.SetCurrentDirectory(@"D:\\Docs");

        path = Path.GetFullPath(@"D:\\FY2018");
        Console.WriteLine($"'D:\\FY2018' resolves to {path}");
        path = Path.GetFullPath(@"D:FY2018");

        // This will be "D:\\Docs\\FY2018" as it happens to match the drive of
        // the current directory
        Console.WriteLine($"'D:FY2018' resolves to {path}");
```

```

Console.WriteLine("Setting current directory to 'C:\\\'");
Directory.SetCurrentDirectory(@"C:\\");

path = Path.GetFullPath(@"D:\\FY2018");
Console.WriteLine($"'D:\\FY2018' resolves to {path}");

// This will be either "D:\\FY2018" or "D:\\FY2018\\FY2018" in the
// subprocess. In the sub process,
// the command prompt set the current directory before launch of our
application, which
// sets a hidden environment variable that is considered.
path = Path.GetFullPath(@"D:\\FY2018");
Console.WriteLine($"'D:\\FY2018' resolves to {path}");

if (args.Length < 1)
{
    Console.WriteLine(@"Launching again, after setting current
directory to D:\\FY2018");
    Uri currentExe = new
Uri(Assembly.GetExecutingAssembly().GetName().CodeBase, UriKind.Absolute);
    string commandLine = $"{"/C cd D:\\FY2018 & \"}
{currentExe.LocalPath}\\ stop";
    ProcessStartInfo psi = new ProcessStartInfo("cmd", commandLine); ;
    Process.Start(psi).WaitForExit();

    Console.WriteLine("Sub process returned:");
    path = Path.GetFullPath(@"D:\\FY2018");
    Console.WriteLine($"'D:\\FY2018' resolves to {path}");
    path = Path.GetFullPath(@"D:\\FY2018");
    Console.WriteLine($"'D:\\FY2018' resolves to {path}");
}
Console.WriteLine("Press any key to continue... ");
Console.ReadKey();
}

}

// The example displays the following output:
//      Current directory is 'C:\\Programs\\file-paths'
//      Setting current directory to 'C:\\'
//      'D:\\FY2018' resolves to D:\\FY2018
//      'D:\\FY2018' resolves to d:\\FY2018
//      Setting current directory to 'D:\\Docs'
//      'D:\\FY2018' resolves to D:\\FY2018
//      'D:\\FY2018' resolves to D:\\Docs\\FY2018
//      Setting current directory to 'C:\\'
//      'D:\\FY2018' resolves to D:\\FY2018
//      'D:\\FY2018' resolves to d:\\FY2018
//      Launching again, after setting current directory to D:\\FY2018
//      Sub process returned:
//      'D:\\FY2018' resolves to D:\\FY2018
//      'D:\\FY2018' resolves to d:\\FY2018
// The subprocess displays the following output:
//      Current directory is 'C:\\'
//      Setting current directory to 'C:\\'
//      'D:\\FY2018' resolves to D:\\FY2018

```

```
//      'D:FY2018' resolves to D:\FY2018\FY2018
//      Setting current directory to 'D:\Docs'
//      'D:\FY2018' resolves to D:\FY2018
//      'D:FY2018' resolves to D:\Docs\FY2018
//      Setting current directory to 'C:\'
//      'D:\FY2018' resolves to D:\FY2018
//      'D:FY2018' resolves to D:\FY2018\FY2018
```

UNC paths

Universal naming convention (UNC) paths, which are used to access network resources, have the following format:

- A server or host name, which is prefaced by `\\"`. The server name can be a NetBIOS machine name or an IP/FQDN address (IPv4 as well as v6 are supported).
- A share name, which is separated from the host name by `\`. Together, the server and share name make up the volume.
- A directory name. The [directory separator character](#) separates subdirectories within the nested directory hierarchy.
- An optional filename. The [directory separator character](#) separates the file path and the filename.

The following are some examples of UNC paths:

| Path | Description |
|--|---|
| <code>\system07\C\$\</code> | The root directory of the <code>c:</code> drive on <code>system07</code> . |
| <code>\Server2\Share\Test\Foo.txt</code> | The <code>Foo.txt</code> file in the <code>Test</code> directory of the <code>\Server2\Share</code> volume. |

UNC paths must always be fully qualified. They can include relative directory segments (`.` and `..`), but these must be part of a fully qualified path. You can use relative paths only by mapping a UNC path to a drive letter.

DOS device paths

The Windows operating system has a unified object model that points to all resources, including files. These object paths are accessible from the console window and are exposed to the Win32 layer through a special folder of symbolic links that legacy DOS and UNC paths are mapped to. This special folder is accessed via the DOS device path syntax, which is one of:

```
\.\.\C:\Test\Foo.txt \?\C:\Test\Foo.txt
```

In addition to identifying a drive by its drive letter, you can identify a volume by using its volume GUID. This takes the form:

```
\.\.\Volume{b75e2c83-0000-0000-0000-602f00000000}\Test\Foo.txt \?\Volume{b75e2c83-0000-0000-0000-602f00000000}\Test\Foo.txt
```

ⓘ Note

DOS device path syntax is supported on .NET implementations running on Windows starting with .NET Core 1.1 and .NET Framework 4.6.2.

The DOS device path consists of the following components:

- The device path specifier (`\.\.` or `\?\`), which identifies the path as a DOS device path.

ⓘ Note

The `\?\` is supported in all versions of .NET Core and .NET 5+ and in .NET Framework starting with version 4.6.2.

- A symbolic link to the "real" device object (C: in the case of a drive name, or `Volume{b75e2c83-0000-0000-0000-602f00000000}` in the case of a volume GUID).

The first segment of the DOS device path after the device path specifier identifies the volume or drive. (For example, `\?\C:\` and `\.\.\BootPartition\.`)

There is a specific link for UNC's that is called, not surprisingly, `UNC`. For example:

```
\.\.\UNC\Server\Share\Test\Foo.txt \?\UNC\Server\Share\Test\Foo.txt
```

For device UNC's, the server/share portion forms the volume. For example, in `\?\server1\utilities\filecomparer\`, the server/share portion is `server1\utilities`. This is significant when calling a method such as `Path.GetFullPath(String, String)` with relative directory segments; it is never possible to navigate past the volume.

DOS device paths are fully qualified by definition and cannot begin with a relative directory segment (`.` or `..`). Current directories never enter into their usage.

Example: Ways to refer to the same file

The following example illustrates some of the ways in which you can refer to a file when using the APIs in the `System.IO` namespace. The example instantiates a `FileInfo` object and uses its `Name` and `Length` properties to display the filename and the length of the file.

C#

```
using System;
using System.IO;

class Program
{
    static void Main()
    {
        string[] filenames = {
            @"c:\temp\test-file.txt",
            @"\127.0.0.1\c$\temp\test-file.txt",
            @"\LOCALHOST\c$\temp\test-file.txt",
            @"\.\c:\temp\test-file.txt",
            @"\?\c:\temp\test-file.txt",
            @"\UNC\LOCALHOST\c$\temp\test-file.txt",
            @"\127.0.0.1\c$\temp\test-file.txt" };

        foreach (var filename in filenames)
        {
            FileInfo fi = new FileInfo(filename);
            Console.WriteLine($"file {fi.Name}: {fi.Length:N0} bytes");
        }
    }
}

// The example displays output like the following:
//     file test-file.txt: 22 bytes
//     file test-file.txt: 22 bytes
//     file test-file.txt: 22 bytes
//     file test-file.txt: 22 bytes
//     file test-file.txt: 22 bytes
//     file test-file.txt: 22 bytes
//     file test-file.txt: 22 bytes
```

Path normalization

Almost all paths passed to Windows APIs are normalized. During normalization, Windows performs the following steps:

- Identifies the path.
- Applies the current directory to partially qualified (relative) paths.
- Canonicalizes component and directory separators.

- Evaluates relative directory components (`.` for the current directory and `..` for the parent directory).
- Trims certain characters.

This normalization happens implicitly, but you can do it explicitly by calling the [Path.GetFullPath](#) method, which wraps a call to the [GetFullPathName\(\) function](#). You can also call the Windows [GetFullPathName\(\) function](#) directly using P/Invoke.

Identify the path

The first step in path normalization is identifying the type of path. Paths fall into one of a few categories:

- They are device paths; that is, they begin with two separators and a question mark or period (`\?\?` or `\?\.`).
- They are UNC paths; that is, they begin with two separators without a question mark or period.
- They are fully qualified DOS paths; that is, they begin with a drive letter, a volume separator, and a component separator (`C:\`).
- They designate a legacy device (`CON`, `LPT1`).
- They are relative to the root of the current drive; that is, they begin with a single component separator (`\`).
- They are relative to the current directory of a specified drive; that is, they begin with a drive letter, a volume separator, and no component separator (`C:\`).
- They are relative to the current directory; that is, they begin with anything else (`temp\testfile.txt`).

The type of the path determines whether or not a current directory is applied in some way. It also determines what the "root" of the path is.

Handle legacy devices

If the path is a legacy DOS device such as `CON`, `COM1`, or `LPT1`, it is converted into a device path by prepending `\?\.\` and returned.

A path that begins with a legacy device name is always interpreted as a legacy device by the [Path.GetFullPath\(String\)](#) method. For example, the DOS device path for `CON.TXT` is `\?\.\CON`, and the DOS device path for `COM1.TXT\file1.txt` is `\?\.\COM1`.

Apply the current directory

If a path isn't fully qualified, Windows applies the current directory to it. UNC paths and device paths do not have the current directory applied. Neither does a full drive with separator `C:\`.

If the path starts with a single component separator, the drive from the current directory is applied. For example, if the file path is `\utilities` and the current directory is `C:\temp\`, normalization produces `C:\utilities`.

If the path starts with a drive letter, volume separator, and no component separator, the last current directory set from the command shell for the specified drive is applied. If the last current directory was not set, the drive alone is applied. For example, if the file path is `D:sources`, the current directory is `C:\Documents\`, and the last current directory on drive D: was `D:\sources\`, the result is `D:\sources\sources`. These "drive relative" paths are a common source of program and script logic errors. Assuming that a path beginning with a letter and a colon isn't relative is obviously not correct.

If the path starts with something other than a separator, the current drive and current directory are applied. For example, if the path is `filecompare` and the current directory is `C:\utilities\`, the result is `C:\utilities\filecompare\`.

Important

Relative paths are dangerous in multithreaded applications (that is, most applications) because the current directory is a per-process setting. Any thread can change the current directory at any time. Starting with .NET Core 2.1, you can call the [Path.GetFullPath\(String, String\)](#) method to get an absolute path from a relative path and the base path (the current directory) that you want to resolve it against.

Canonicalize separators

All forward slashes (`/`) are converted into the standard Windows separator, the back slash (`\`). If they are present, a series of slashes that follow the first two slashes are collapsed into a single slash.

Evaluate relative components

As the path is processed, any components or segments that are composed of a single or a double period (`.` or `..`) are evaluated:

- For a single period, the current segment is removed, since it refers to the current directory.
- For a double period, the current segment and the parent segment are removed, since the double period refers to the parent directory.

Parent directories are only removed if they aren't past the root of the path. The root of the path depends on the type of path. It is the drive (`c:\`) for DOS paths, the server/share for UNC's (`\Server\Share`), and the device path prefix for device paths (`\?\` or `\\.\).`

Trim characters

Along with the runs of separators and relative segments removed earlier, some additional characters are removed during normalization:

- If a segment ends in a single period, that period is removed. (A segment of a single or double period is normalized in the previous step. A segment of three or more periods is not normalized and is actually a valid file/directory name.)
- If the path doesn't end in a separator, all trailing periods and spaces (U+0020) are removed. If the last segment is simply a single or double period, it falls under the relative components rule above.

This rule means that you can create a directory name with a trailing space by adding a trailing separator after the space.

Important

You should **never** create a directory or filename with a trailing space. Trailing spaces can make it difficult or impossible to access a directory, and applications commonly fail when attempting to handle directories or files whose names include trailing spaces.

Skip normalization

Normally, any path passed to a Windows API is (effectively) passed to the [GetFullPathName function](#) and normalized. There is one important exception: a device path that begins with a question mark instead of a period. Unless the path starts exactly with `\?\` (note the use of the canonical backslash), it is normalized.

Why would you want to skip normalization? There are three major reasons:

1. To get access to paths that are normally unavailable but are legal. A file or directory called `hidden.`, for example, is impossible to access in any other way.
2. To improve performance by skipping normalization if you've already normalized.
3. On .NET Framework only, to skip the `MAX_PATH` check for path length to allow for paths that are greater than 259 characters. Most APIs allow this, with some exceptions.

ⓘ Note

.NET Core and .NET 5+ handles long paths implicitly and does not perform a `MAX_PATH` check. The `MAX_PATH` check applies only to .NET Framework.

Skipping normalization and max path checks is the only difference between the two device path syntaxes; they are otherwise identical. Be careful with skipping normalization, since you can easily create paths that are difficult for "normal" applications to deal with.

Paths that start with `\?\` are still normalized if you explicitly pass them to the [GetFullPathName function](#).

You can pass paths of more than `MAX_PATH` characters to [GetFullPathName](#) without `\?\`. It supports arbitrary length paths up to the maximum string size that Windows can handle.

Case and the Windows file system

A peculiarity of the Windows file system that non-Windows users and developers find confusing is that path and directory names are case-insensitive. That is, directory and file names reflect the casing of the strings used when they are created. For example, the method call

C#

```
Directory.CreateDirectory("TeStDiReCtOrY");
```

creates a directory named `TeStDiReCtOrY`. If you rename a directory or file to change its case, the directory or file name reflects the case of the string used when you rename it. For example, the following code renames a file named `test.txt` to `Test.txt`:

C#

```
using System.IO;

class Example
{
    static void Main()
    {
        var fi = new FileInfo(@".\test.txt");
        fi.MoveTo(@".\Test.txt");
    }
}
```

However, directory and file name comparisons are case-insensitive. If you search for a file named "test.txt", .NET file system APIs ignore case in the comparison. "Test.txt", "TEST.TXT", "test.TXT", and any other combination of uppercase and lowercase letters will match "test.txt".

 Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

Common I/O Tasks

Article • 09/15/2021

The [System.IO](#) namespace provides several classes that allow for various actions, such as reading and writing, to be performed on files, directories, and streams. For more information, see [File and Stream I/O](#).

Common File Tasks

| To do this... | See the example in this topic... |
|------------------------------|--|
| Create a text file | File.CreateText method FileInfo.CreateText method File.Create method FileInfo.Create method |
| Write to a text file | How to: Write Text to a File How to: Write a Text File (C++/CLI) |
| Read from a text file | How to: Read Text from a File |
| Append text to a file | How to: Open and Append to a Log File File.AppendText method FileInfo.AppendText method |
| Rename or move a file | File.Move method FileInfo.MoveTo method |
| Delete a file | File.Delete method FileInfo.Delete method |
| Copy a file | File.Copy method FileInfo.CopyTo method |
| Get the size of a file | FileInfo.Length property |
| Get the attributes of a file | File.GetAttributes method |

| To do this... | See the example in this topic... |
|--|---|
| Set the attributes of a file | File.SetAttributes method |
| Determine whether a file exists | File.Exists method |
| Read from a binary file | How to: Read and Write to a Newly Created Data File |
| Write to a binary file | How to: Read and Write to a Newly Created Data File |
| Retrieve a file name extension | Path.GetExtension method |
| Retrieve the fully qualified path of a file | Path.GetFullPath method |
| Retrieve the file name and extension from a path | Path.GetFileName method |
| Change the extension of a file | Path.ChangeExtension method |

Common Directory Tasks

| To do this... | See the example in this topic... |
|--|---|
| Access a file in a special folder such as My Documents | How to: Write Text to a File |
| Create a directory | Directory.CreateDirectory method |
| | FileInfo.Directory property |
| Create a subdirectory | DirectoryInfo.CreateSubdirectory method |
| Rename or move a directory | Directory.Move method |
| | DirectoryInfo.MoveTo method |
| Copy a directory | How to: Copy Directories |
| Delete a directory | Directory.Delete method |
| | DirectoryInfo.Delete method |
| See the files and subdirectories in a directory | How to: Enumerate Directories and Files |
| Find the size of a directory | System.IO.Directory class |
| Determine whether a directory exists | Directory.Exists method |

See also

- [File and Stream I/O](#)
- [Composing Streams](#)
- [Asynchronous File I/O](#)

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

How to: Copy directories

Article • 12/14/2022

This article demonstrates how to use I/O classes to synchronously copy the contents of a directory to another location.

For an example of asynchronous file copy, see [Asynchronous file I/O](#).

This example copies subdirectories by setting the `recursive` parameter of the `CopyDirectory` method to `true`. The `CopyDirectory` method recursively copies subdirectories by calling itself on each subdirectory until there are no more to copy.

Example

C#

```
using System.IO;

CopyDirectory(@".\", @".\copytest", true);

static void CopyDirectory(string sourceDir, string destinationDir, bool
recursive)
{
    // Get information about the source directory
    var dir = new DirectoryInfo(sourceDir);

    // Check if the source directory exists
    if (!dir.Exists)
        throw new DirectoryNotFoundException($"Source directory not found:
{dir.FullName}");

    // Cache directories before we start copying
    DirectoryInfo[] dirs = dir.GetDirectories();

    // Create the destination directory
    Directory.CreateDirectory(destinationDir);

    // Get the files in the source directory and copy to the destination
    // directory
    foreach (FileInfo file in dir.GetFiles())
    {
        string targetFilePath = Path.Combine(destinationDir, file.Name);
        file.CopyTo(targetFilePath);
    }

    // If recursive and copying subdirectories, recursively call this method
    if (recursive)
    {
```

```
        foreach ( DirectoryInfo subDir in dirs )
    {
        string newDestinationDir = Path.Combine(destinationDir,
subDir.Name);
        CopyDirectory(subDir.FullName, newDestinationDir, true);
    }
}
```

See also

- [FileInfo](#)
- [DirectoryInfo](#)
- [FileStream](#)
- [File and stream I/O](#)
- [Common I/O tasks](#)
- [Asynchronous file I/O](#)

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

How to: Enumerate directories and files

Article • 09/15/2021

Enumerable collections provide better performance than arrays when you work with large collections of directories and files. To enumerate directories and files, use methods that return an enumerable collection of directory or file names, or their [DirectoryInfo](#), [FileInfo](#), or [FileSystemInfo](#) objects.

If you want to search and return only the names of directories or files, use the enumeration methods of the [Directory](#) class. If you want to search and return other properties of directories or files, use the [DirectoryInfo](#) and [FileSystemInfo](#) classes.

You can use enumerable collections from these methods as the [IEnumerable<T>](#) parameter for constructors of collection classes like [List<T>](#).

The following table summarizes the methods that return enumerable collections of files and directories:

| To search and return | Use method |
|--|--|
| Directory names | Directory.EnumerateDirectories |
| Directory information (DirectoryInfo) | DirectoryInfo.EnumerateDirectories |
| File names | Directory.EnumerateFiles |
| File information (FileInfo) | DirectoryInfo.EnumerateFiles |
| File system entry names | Directory.EnumerateFileSystemEntries |
| File system entry information (FileSystemInfo) | DirectoryInfo.EnumerateFileSystemInfos |
| Directory and file names | Directory.EnumerateFileSystemEntries |

ⓘ Note

Although you can immediately enumerate all the files in the subdirectories of a parent directory by using the [AllDirectories](#) option of the optional [SearchOption](#) enumeration, [UnauthorizedAccessException](#) errors may make the enumeration incomplete. You can catch these exceptions by first enumerating directories and then enumerating files.

Examples: Use the [Directory](#) class

The following example uses the [Directory.EnumerateDirectories\(String\)](#) method to get a list of the top-level directory names in a specified path.

```
C#  
  
using System;  
using System.Collections.Generic;  
using System.IO;  
  
class Program  
{  
    private static void Main(string[] args)  
    {  
        try  
        {  
            // Set a variable to the My Documents path.  
            string docPath =  
                Environment.GetFolderPath(Environment.SpecialFolder.MyDocuments);  
  
            List<string> dirs = new List<string>  
(Directory.EnumerateDirectories(docPath));  
  
            foreach (var dir in dirs)  
            {  
                Console.WriteLine($"  
{dir.Substring(dir.LastIndexOf(Path.DirectorySeparatorChar) + 1)}");  
            }  
            Console.WriteLine($"{dirs.Count} directories found.");  
        }  
        catch (UnauthorizedAccessException ex)  
        {  
            Console.WriteLine(ex.Message);  
        }  
        catch (PathTooLongException ex)  
        {  
            Console.WriteLine(ex.Message);  
        }  
    }  
}
```

The following example uses the [Directory.EnumerateFiles\(String, String, SearchOption\)](#) method to recursively enumerate all file names in a directory and subdirectories that match a certain pattern. It then reads each line of each file and displays the lines that contain a specified string, with their filenames and paths.

```
C#  
  
using System;  
using System.IO;  
using System.Linq;
```

```

class Program
{
    static void Main(string[] args)
    {
        try
        {
            // Set a variable to the My Documents path.
            string docPath =
                Environment.GetFolderPath(Environment.SpecialFolder.MyDocuments);

            var files = from file in Directory.EnumerateFiles(docPath,
                "*.txt", SearchOption.AllDirectories)
                from line in File.ReadLines(file)
                where line.Contains("Microsoft")
                select new
                {
                    File = file,
                    Line = line
                };

            foreach (var f in files)
            {
                Console.WriteLine($"{f.File}\t{f.Line}");
            }
            Console.WriteLine($"{files.Count()} files found.");
        }
        catch (UnauthorizedAccessException uAEx)
        {
            Console.WriteLine(uAEx.Message);
        }
        catch (PathTooLongException pathEx)
        {
            Console.WriteLine(pathEx.Message);
        }
    }
}

```

Examples: Use the DirectoryInfo class

The following example uses the [DirectoryInfo.EnumerateDirectories](#) method to list a collection of top-level directories whose [CreationTimeUtc](#) is earlier than a certain [DateTime](#) value.

C#

```

using System;
using System.IO;

namespace EnumDir
{

```

```

class Program
{
    static void Main(string[] args)
    {
        // Set a variable to the Documents path.
        string docPath =
Environment.GetFolderPath(Environment.SpecialFolder.MyDocuments);

        DirectoryInfo dirPrograms = new DirectoryInfo(docPath);
        DateTime StartOf2009 = new DateTime(2009, 01, 01);

        var dirs = from dir in dirPrograms.EnumerateDirectories()
                   where dir.CreationTimeUtc > StartOf2009
                   select new
        {
            ProgDir = dir,
        };

        foreach (var di in dirs)
        {
            Console.WriteLine($"{di.ProgDir.Name}");
        }
    }
}
// </Snippet1>

```

The following example uses the [DirectoryInfo.EnumerateFiles](#) method to list all files whose [Length](#) exceeds 10MB. This example first enumerates the top-level directories, to catch possible unauthorized access exceptions, and then enumerates the files.

C#

```

using System;
using System.IO;

class Program
{
    static void Main(string[] args)
    {
        // Set a variable to the My Documents path.
        string docPath =
Environment.GetFolderPath(Environment.SpecialFolder.MyDocuments);

        DirectoryInfo diTop = new DirectoryInfo(docPath);

        try
        {
            foreach (var fi in diTop.EnumerateFiles())
            {
                try
                {

```

```
// Display each file over 10 MB;
if (fi.Length > 10000000)
{
    Console.WriteLine(${fi.FullName}\t\t{fi.Length.ToString("N0")}");
}
catch (UnauthorizedAccessException unAuthTop)
{
    Console.WriteLine(${unAuthTop.Message});
}
}

foreach (var di in diTop.EnumerateDirectories("*"))
{
    try
    {
        foreach (var fi in di.EnumerateFiles("*",
SearchOption.AllDirectories))
        {
            try
            {
                // Display each file over 10 MB;
                if (fi.Length > 10000000)
                {
                    Console.WriteLine(${fi.FullName}\t\t{fi.Length.ToString("N0")});
                }
            }
            catch (UnauthorizedAccessException unAuthFile)
            {
                Console.WriteLine(${unAuthFile:
{unAuthFile.Message}});
            }
        }
    }
    catch (UnauthorizedAccessException unAuthSubDir)
    {
        Console.WriteLine(${unAuthSubDir:
{unAuthSubDir.Message}});
    }
}
}

catch (FileNotFoundException dirNotFound)
{
    Console.WriteLine(${dirNotFound.Message});
}
catch (UnauthorizedAccessException unAuthDir)
{
    Console.WriteLine(${unAuthDir: {unAuthDir.Message}});
}
catch (PathTooLongException longPath)
{
    Console.WriteLine(${longPath.Message});
}
```

```
    }  
}
```

See also

- [File and stream I/O](#)

 [Collaborate with us on GitHub](#)

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

[.NET feedback](#)

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

How to: Read and write to a newly created data file

Article • 09/15/2021

The [System.IO.BinaryWriter](#) and [System.IO.BinaryReader](#) classes are used for writing and reading data other than character strings. The following example shows how to create an empty file stream, write data to it, and read data from it.

The example creates a data file called *Test.data* in the current directory, creates the associated [BinaryWriter](#) and [BinaryReader](#) objects, and uses the [BinaryWriter](#) object to write the integers 0 through 10 to *Test.data*, which leaves the file pointer at the end of the file. The [BinaryReader](#) object then sets the file pointer back to the origin and reads out the specified content.

ⓘ Note

If *Test.data* already exists in the current directory, an [IOException](#) exception is thrown. Use the file mode option [FileMode.Create](#) rather than [FileMode.CreateNew](#) to always create a new file without throwing an exception.

Example

C#

```
using System;
using System.IO;

class MyStream
{
    private const string FILE_NAME = "Test.data";

    public static void Main()
    {
        if (File.Exists(FILE_NAME))
        {
            Console.WriteLine($"{FILE_NAME} already exists!");
            return;
        }

        using (FileStream fs = new FileStream(FILE_NAME,
FileMode.CreateNew))
        {
            using (BinaryWriter w = new BinaryWriter(fs))

```

```

    {
        for (int i = 0; i < 11; i++)
        {
            w.Write(i);
        }
    }

    using (FileStream fs = new FileStream(FILE_NAME, FileMode.Open,
FileAccess.Read))
    {
        using (BinaryReader r = new BinaryReader(fs))
        {
            for (int i = 0; i < 11; i++)
            {
                Console.WriteLine(r.ReadInt32());
            }
        }
    }
}

// The example creates a file named "Test.data" and writes the integers 0
// through 10 to it in binary format.
// It then writes the contents of Test.data to the console with each integer
// on a separate line.

```

See also

- [BinaryReader](#)
- [BinaryWriter](#)
- [FileStream](#)
- [FileStream.Seek](#)
- [SeekOrigin](#)
- [How to: Enumerate directories and files](#)
- [How to: Open and append to a log file](#)
- [How to: Read text from a file](#)
- [How to: Write text to a file](#)
- [How to: Read characters from a string](#)
- [How to: Write characters to a string](#)
- [File and stream I/O](#)



[Collaborate with us on](#)

GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

How to: Open and append to a log file

Article • 09/15/2021

[StreamWriter](#) and [StreamReader](#) write characters to and read characters from streams. The following code example opens the *log.txt* file for input, or creates it if it doesn't exist, and appends log information to the end of the file. The example then writes the contents of the file to standard output for display.

As an alternative to this example, you could store the information as a single string or string array, and use the [File.WriteAllText](#) or [File.WriteAllLines](#) method to achieve the same functionality.

ⓘ Note

Visual Basic users may choose to use the methods and properties provided by the [Log](#) class or [FileSystem](#) class for creating or writing to log files.

Example

C#

```
using System;
using System.IO;

class DirAppend
{
    public static void Main()
    {
        using (StreamWriter w = File.AppendText("log.txt"))
        {
            Log("Test1", w);
            Log("Test2", w);
        }

        using (StreamReader r = File.OpenText("log.txt"))
        {
            DumpLog(r);
        }
    }

    public static void Log(string logMessage, TextWriter w)
    {
        w.WriteLine("\r\nLog Entry : ");
        w.WriteLine($"{DateTime.Now.ToString("yyyy-MM-dd HH:mm:ss.fff")}");
        w.WriteLine($"{DateTime.Now.ToString("HH:mm:ss.fff")}");
        w.WriteLine(" :");
    }
}
```

```

        w.WriteLine($" :{logMessage}");
        w.WriteLine ("-----");
    }

    public static void DumpLog(StreamReader r)
    {
        string line;
        while ((line = r.ReadLine()) != null)
        {
            Console.WriteLine(line);
        }
    }
}

// The example creates a file named "log.txt" and writes the following lines
// to it,
// or appends them to the existing "log.txt" file:

// Log Entry : <current long time string> <current long date string>
// :
// :Test1
// ----

// Log Entry : <current long time string> <current long date string>
// :
// :Test2
// ----

// It then writes the contents of "log.txt" to the console.

```

See also

- [StreamWriter](#)
- [StreamReader](#)
- [File.AppendText](#)
- [File.OpenText](#)
- [StreamReader.ReadLine](#)
- [How to: Enumerate directories and files](#)
- [How to: Read and write to a newly created data file](#)
- [How to: Read text from a file](#)
- [How to: Write text to a file](#)
- [How to: Read characters from a string](#)
- [How to: Write characters to a string](#)
- [File and stream I/O](#)

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

How to: Write text to a file

Article • 12/14/2022

This article shows different ways to write text to a file for a .NET app.

The following classes and methods are typically used to write text to a file:

- [StreamWriter](#) contains methods to write to a file synchronously ([Write](#) and [WriteLine](#)) or asynchronously ([WriteAsync](#) and [WriteLineAsync](#)).
- [File](#) provides static methods to write text to a file such as [WriteAllLines](#) and [WriteAllText](#), or to append text to a file such as [AppendAllLines](#), [AppendAllText](#), and [AppendText](#).
- [Path](#) is for strings that have file or directory path information. It contains the [Combine](#) method and in .NET Core 2.1 and later, the [Join](#) and [TryJoin](#) methods. These methods let you concatenate strings for building a file or directory path.

ⓘ Note

The following examples show only the minimum amount of code needed. A real-world app usually provides more robust error checking and exception handling.

Example: Synchronously write text with StreamWriter

The following example shows how to use the [StreamWriter](#) class to synchronously write text to a new file one line at a time. Because the [StreamWriter](#) object is declared and instantiated in a `using` statement, the [Dispose](#) method is invoked, which automatically flushes and closes the stream.

C#

```
using System;
using System.IO;

class Program
{
    static void Main(string[] args)
    {

        // Create a string array with the lines of text
        string[] lines = { "First line", "Second line", "Third line" };
    }
}
```

```

// Set a variable to the Documents path.
string docPath =
    Environment.GetFolderPath(Environment.SpecialFolder.MyDocuments);

// Write the string array to a new file named "WriteLines.txt".
using (StreamWriter outputFile = new
StreamWriter(Path.Combine(docPath, "WriteLines.txt")))
{
    foreach (string line in lines)
        outputFile.WriteLine(line);
}
}

// The example creates a file named "WriteLines.txt" with the following
contents:
// First line
// Second line
// Third line

```

Example: Synchronously append text with StreamWriter

The following example shows how to use the [StreamWriter](#) class to synchronously append text to the text file created in the first example:

C#

```

using System;
using System.IO;

class Program
{
    static void Main(string[] args)
    {

        // Set a variable to the Documents path.
        string docPath =
Environment.GetFolderPath(Environment.SpecialFolder.MyDocuments);

        // Append text to an existing file named "WriteLines.txt".
        using (StreamWriter outputFile = new
StreamWriter(Path.Combine(docPath, "WriteLines.txt"), true))
        {
            outputFile.WriteLine("Fourth Line");
        }
    }
}

```

```
// The example adds the following line to the contents of "WriteLines.txt":  
// Fourth Line
```

Example: Asynchronously write text with StreamWriter

The following example shows how to asynchronously write text to a new file using the [StreamWriter](#) class. To invoke the [WriteAsync](#) method, the method call must be within an `async` method.

C#

```
using System;  
using System.IO;  
using System.Threading.Tasks;  
  
class Program  
{  
    static async Task Main()  
    {  
        // Set a variable to the Documents path.  
        string docPath =  
Environment.GetFolderPath(Environment.SpecialFolder.MyDocuments);  
  
        // Write the specified text asynchronously to a new file named  
        // "WriteTextAsync.txt".  
        using (StreamWriter outputFile = new  
StreamWriter(Path.Combine(docPath, "WriteTextAsync.txt")))  
        {  
            await outputFile.WriteAsync("This is a sentence.");  
        }  
    }  
    // The example creates a file named "WriteTextAsync.txt" with the following  
    // contents:  
    // This is a sentence.
```

Example: Write and append text with the File class

The following example shows how to write text to a new file and append new lines of text to the same file using the [File](#) class. The [WriteAllText](#) and [AppendAllLines](#) methods open and close the file automatically. If the path you provide to the [WriteAllText](#) method already exists, the file is overwritten.

C#

```
using System;
using System.IO;

class Program
{
    static void Main(string[] args)
    {
        // Create a string with a line of text.
        string text = "First line" + Environment.NewLine;

        // Set a variable to the Documents path.
        string docPath =
Environment.GetFolderPath(Environment.SpecialFolder.MyDocuments);

        // Write the text to a new file named "WriteFile.txt".
        File.WriteAllText(Path.Combine(docPath, "WriteFile.txt"), text);

        // Create a string array with the additional lines of text
        string[] lines = { "New line 1", "New line 2" };

        // Append new lines of text to the file
        File.AppendAllLines(Path.Combine(docPath, "WriteFile.txt"), lines);
    }
}

// The example creates a file named "WriteFile.txt" with the contents:
// First line
// And then appends the following contents:
// New line 1
// New line 2
```

See also

- [StreamWriter](#)
- [Path](#)
- [File.CreateText](#)
- [How to: Enumerate directories and files](#)
- [How to: Read and write to a newly created data file](#)
- [How to: Open and append to a log file](#)
- [How to: Read text from a file](#)
- [File and stream I/O](#)



Collaborate with us on
GitHub

.NET

.NET feedback

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

How to: Read text from a file

Article • 10/04/2022

The following examples show how to read text synchronously and asynchronously from a text file using .NET for desktop apps. In both examples, when you create the instance of the [StreamReader](#) class, you provide the relative or absolute path to the file.

ⓘ Note

These code examples don't apply to Universal Windows (UWP) apps because the Windows runtime provides different stream types for reading and writing to files. For an example that shows how to read text from a file in a UWP app, see [Quickstart: Reading and writing files](#). For examples that show how to convert between .NET Framework streams and Windows Runtime streams, see [How to: Convert between .NET Framework streams and Windows Runtime streams](#).

Example: Synchronous read in a console app

The following example shows a synchronous read operation within a console app. This example opens the text file using a stream reader, copies the contents to a string, and outputs the string to the console.

ⓘ Important

The example assumes that a file named *TestFile.txt* already exists in the same folder as the app.

C#

```
using System;
using System.IO;

class Program
{
    public static void Main()
    {
        try
        {
            // Open the text file using a stream reader.
            using (var sr = new StreamReader("TestFile.txt"))
            {
                // Read the stream as a string, and write the string to the
                // console.
                string contents = sr.ReadToEnd();
                Console.WriteLine(contents);
            }
        }
    }
}
```

```
        Console.WriteLine(sr.ReadToEnd());
    }
}
catch (IOException e)
{
    Console.WriteLine("The file could not be read:");
    Console.WriteLine(e.Message);
}
}
}
```

Example: Asynchronous read in a WPF app

The following example shows an asynchronous read operation in a Windows Presentation Foundation (WPF) app.

ⓘ Important

The example assumes that a file named *TestFile.txt* already exists in the same folder as the app.

C#

```
using System.IO;
using System.Windows;

namespace TextFiles;

/// <summary>
/// Interaction logic for MainWindow.xaml
/// </summary>
public partial class MainWindow : Window
{
    public MainWindow() => InitializeComponent();

    private async void MainWindow_Loaded(object sender, RoutedEventArgs e)
    {
        try
        {
            using (var sr = new StreamReader("TestFile.txt"))
            {
                ResultBlock.Text = await sr.ReadToEndAsync();
            }
        }
        catch (FileNotFoundException ex)
        {
            ResultBlock.Text = ex.Message;
        }
    }
}
```

```
    }  
}
```

See also

- [StreamReader](#)
- [File.OpenText](#)
- [StreamReader.ReadLine](#)
- [Asynchronous file I/O](#)
- [How to: Create a directory listing](#)
- [Quickstart: Reading and writing files](#)
- [How to: Convert between .NET Framework streams and Windows Runtime streams](#)
- [How to: Read and write to a newly created data file](#)
- [How to: Open and append to a log file](#)
- [How to: Write text to a file](#)
- [How to: Read characters from a string](#)
- [How to: Write characters to a string](#)
- [File and stream I/O](#)

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback [here](#).

 [Open a documentation issue](#)

 [Provide product feedback](#)

How to: Read characters from a string

Article • 09/15/2021

The following code examples show how to read characters synchronously or asynchronously from a string.

Example: Read characters synchronously

This example reads 13 characters synchronously from a string, stores them in an array, and displays them. The example then reads the rest of the characters in the string, stores them in the array starting at the sixth element, and displays the contents of the array.

C#

```
using System;
using System.IO;

public class CharsFromStr
{
    public static void Main()
    {
        string str = "Some number of characters";
        char[] b = new char[str.Length];

        using (StringReader sr = new StringReader(str))
        {
            // Read 13 characters from the string into the array.
            sr.Read(b, 0, 13);
            Console.WriteLine(b);

            // Read the rest of the string starting at the current string
            position.
            // Put in the array starting at the 6th array member.
            sr.Read(b, 5, str.Length - 13);
            Console.WriteLine(b);
        }
    }
}

// The example has the following output:
//
// Some number o
// Some f characters
```

Example: Read characters asynchronously

The next example is the code behind a WPF app. On window load, the example asynchronously reads all characters from a [TextBox](#) control and stores them in an array. It then asynchronously writes each letter or white-space character to a separate line of a [TextBlock](#) control.

C#

```
using System;
using System.Text;
using System.Windows;
using System.IO;

namespace StringReaderWriter
{
    /// <summary>
    /// Interaction logic for MainWindow.xaml
    /// </summary>
    public partial class MainWindow : Window
    {
        public MainWindow()
        {
            InitializeComponent();
        }

        private async void Window_Loaded(object sender, RoutedEventArgs e)
        {
            char[] charsRead = new char[UserInput.Text.Length];
            using (StringReader reader = new StringReader(UserInput.Text))
            {
                await reader.ReadAsync(charsRead, 0, UserInput.Text.Length);
            }

            StringBuilder reformattedText = new StringBuilder();
            using (StringWriter writer = new StringWriter(reformattedText))
            {
                foreach (char c in charsRead)
                {
                    if (char.IsLetter(c) || char.IsWhiteSpace(c))
                    {
                        await writer.WriteLineAsync(char.ToLower(c));
                    }
                }
            }
            Result.Text = reformattedText.ToString();
        }
    }
}
```

See also

- [StringReader](#)
- [StringReader.Read](#)
- [Asynchronous file I/O](#)
- [How to: Create a directory listing](#)
- [How to: Read and write to a newly created data file](#)
- [How to: Open and append to a log file](#)
- [How to: Read text from a file](#)
- [How to: Write text to a file](#)
- [How to: Write characters to a string](#)
- [File and stream I/O](#)

 **Collaborate with us on GitHub**

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

How to: Write characters to a string

Article • 09/15/2021

The following code examples write characters synchronously or asynchronously from a character array into a string.

Example: Write characters synchronously in a console app

The following example uses a [StringWriter](#) to write five characters synchronously to a [StringBuilder](#) object.

C#

```
using System;
using System.IO;
using System.Text;

public class CharsToStr
{
    public static void Main()
    {
        StringBuilder sb = new StringBuilder("Start with a string and add
from ");
        char[] b = { 'c', 'h', 'a', 'r', '.', ' ', 'B', 'u', 't', ' ', 'n',
'o', 't', ' ', 'a', 'l', 'l' };

        using (StringWriter sw = new StringWriter(sb))
        {
            // Write five characters from the array into the StringBuilder.
            sw.Write(b, 0, 5);
            Console.WriteLine(sb);
        }
    }
}

// The example has the following output:
//
// Start with a string and add from char.
```

Example: Write characters asynchronously in a WPF app

The next example is the code behind a WPF app. On window load, the example asynchronously reads all characters from a [TextBox](#) control and stores them in an array. It then asynchronously writes each letter or white-space character to a separate line of a [TextBlock](#) control.

C#

```
using System;
using System.Text;
using System.Windows;
using System.IO;

namespace StringReaderWriter
{
    /// <summary>
    /// Interaction logic for MainWindow.xaml
    /// </summary>
    public partial class MainWindow : Window
    {
        public MainWindow()
        {
            InitializeComponent();
        }

        private async void Window_Loaded(object sender, RoutedEventArgs e)
        {
            char[] charsRead = new char[UserInput.Text.Length];
            using (StringReader reader = new StringReader(UserInput.Text))
            {
                await reader.ReadAsync(charsRead, 0, UserInput.Text.Length);
            }

            StringBuilder reformattedText = new StringBuilder();
            using (StringWriter writer = new StringWriter(reformattedText))
            {
                foreach (char c in charsRead)
                {
                    if (char.IsLetter(c) || char.IsWhiteSpace(c))
                    {
                        await writer.WriteLineAsync(char.ToLower(c));
                    }
                }
            }
            Result.Text = reformattedText.ToString();
        }
    }
}
```

See also

- [StringWriter](#)
- [StringWriter.Write](#)
- [StringBuilder](#)
- [File and stream I/O](#)
- [Asynchronous file I/O](#)
- [How to: Enumerate directories and files](#)
- [How to: Read and write to a newly created data file](#)
- [How to: Open and append to a log file](#)
- [How to: Read text from a file](#)
- [How to: Write text to a file](#)
- [How to: Read characters from a string](#)

 **Collaborate with us on GitHub**

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

 .NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

How to: Add or remove Access Control List entries (.NET Framework only)

Article • 09/15/2021

To add or remove Access Control List (ACL) entries to or from a file or directory, get the [FileSecurity](#) or [DirectorySecurity](#) object from the file or directory. Modify the object, and then apply it back to the file or directory.

Add or remove an ACL entry from a file

1. Call the [File.GetAccessControl](#) method to get a [FileSecurity](#) object that contains the current ACL entries of a file.
2. Add or remove ACL entries from the [FileSecurity](#) object returned from step 1.
3. To apply the changes, pass the [FileSecurity](#) object to the [File.SetAccessControl](#) method.

Add or remove an ACL entry from a directory

1. Call the [Directory.GetAccessControl](#) method to get a [DirectorySecurity](#) object that contains the current ACL entries of a directory.
2. Add or remove ACL entries from the [DirectorySecurity](#) object returned from step 1.
3. To apply the changes, pass the [DirectorySecurity](#) object to the [Directory.SetAccessControl](#) method.

Example

You must use a valid user or group account to run this example. The example uses a [File](#) object. Use the same procedure for the [FileInfo](#), [Directory](#), and [DirectoryInfo](#) classes.

C#

```
using System;
using System.IO;
using System.Security.AccessControl;

namespace FileSystemExample
{
    class FileExample
```

```
{  
    public static void Main()  
    {  
        try  
        {  
            string fileName = "test.xml";  
  
            Console.WriteLine("Adding access control entry for "  
                + fileName);  
  
            // Add the access control entry to the file.  
            AddFileSecurity(fileName, @"DomainName\AccountName",  
                FileSystemRights.ReadData, AccessControlType.Allow);  
  
            Console.WriteLine("Removing access control entry from "  
                + fileName);  
  
            // Remove the access control entry from the file.  
            RemoveFileSecurity(fileName, @"DomainName\AccountName",  
                FileSystemRights.ReadData, AccessControlType.Allow);  
  
            Console.WriteLine("Done.");  
        }  
        catch (Exception e)  
        {  
            Console.WriteLine(e);  
        }  
    }  
  
    // Adds an ACL entry on the specified file for the specified  
    account.  
    public static void AddFileSecurity(string fileName, string account,  
        FileSystemRights rights, AccessControlType controlType)  
    {  
  
        // Get a FileSecurity object that represents the  
        // current security settings.  
        FileSecurity fSecurity = File.GetAccessControl(fileName);  
  
        // Add the FileSystemAccessRule to the security settings.  
        fSecurity.AddAccessRule(new FileSystemAccessRule(account,  
            rights, controlType));  
  
        // Set the new access settings.  
        File.SetAccessControl(fileName, fSecurity);  
    }  
  
    // Removes an ACL entry on the specified file for the specified  
    account.  
    public static void RemoveFileSecurity(string fileName, string  
account,  
        FileSystemRights rights, AccessControlType controlType)  
    {  
  
        // Get a FileSecurity object that represents the
```

```
// current security settings.  
FileSecurity fSecurity = File.GetAccessControl(fileName);  
  
// Remove the FileSystemAccessRule from the security settings.  
fSecurity.RemoveAccessRule(new FileSystemAccessRule(account,  
    rights, controlType));  
  
// Set the new access settings.  
File.SetAccessControl(fileName, fSecurity);  
}  
}  
}
```

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

How to: Compress and extract files

Article • 08/12/2022

The [System.IO.Compression](#) namespace contains the following classes for compressing and decompressing files and streams. You also can use these types to read and modify the contents of a compressed file:

- [ZipFile](#)
- [ZipArchive](#)
- [ZipArchiveEntry](#)
- [DeflateStream](#)
- [GZipStream](#)

The following examples show some of the operations you can perform with compressed files. These examples require the following NuGet packages to be added to your project:

- [System.IO.Compression](#) ↗
- [System.IO.Compression.ZipFile](#) ↗

If you're using .NET Framework, add references to these two libraries to your project:

- `System.IO.Compression`
- `System.IO.Compression.FileSystem`

Example 1: Create and extract a .zip file

The following example shows how to create and extract a compressed `.zip` file by using the [ZipFile](#) class. The example compresses the contents of a folder into a new `.zip` file, and then extracts the file to a new folder.

To run the sample, create a *start* folder in your program folder and populate it with files to zip.

C#

```
using System;
using System.IO.Compression;

class Program
{
    static void Main(string[] args)
    {
        string startPath = @".\start";
        string zipPath = @".\result.zip";
```

```
        string extractPath = @".\extract";

        ZipFile.CreateDirectory(startPath, zipPath);

        ZipFile.ExtractToDirectory(zipPath, extractPath);
    }
}
```

Example 2: Extract specific file extensions

The following example iterates through the contents of an existing `.zip` file and extracts files with a `.txt` extension. It uses the [ZipArchive](#) class to access the `.zip` file, and the [ZipArchiveEntry](#) class to inspect the individual entries. The extension method [ExtractToFile](#) for the [ZipArchiveEntry](#) object is available in the [System.IO.Compression.ZipFileExtensions](#) class.

To run the sample, place a `.zip` file called `result.zip` in your program folder. When prompted, provide a folder name to extract to.

ⓘ Important

When unzipping files, you must look for malicious file paths, which can escape from the directory you unzip into. This is known as a path traversal attack. The following example demonstrates how to check for malicious file paths and provides a safe way to unzip.

C#

```
using System;
using System.IO;
using System.IO.Compression;

class Program
{
    static void Main(string[] args)
    {
        string zipPath = @".\result.zip";

        Console.WriteLine("Provide path where to extract the zip file:");
        string extractPath = Console.ReadLine();

        // Normalizes the path.
        extractPath = Path.GetFullPath(extractPath);

        // Ensures that the last character on the extraction path
        // is the directory separator char.
```

```

        // Without this, a malicious zip file could try to traverse outside
        // of the expected
        // extraction path.
        if (!extractPath.EndsWith(Path.DirectorySeparatorChar.ToString(),
StringComparison.OrdinalIgnoreCase))
            extractPath += Path.DirectorySeparatorChar;

        using (ZipArchive archive = ZipFile.OpenRead(zipPath))
        {
            foreach (ZipArchiveEntry entry in archive.Entries)
            {
                if (entry.FullName.EndsWith(".txt",
StringComparison.OrdinalIgnoreCase))
                {
                    // Gets the full path to ensure that relative segments
                    // are removed.
                    string destinationPath =
Path.GetFullPath(Path.Combine(extractPath, entry.FullName));

                    // Ordinal match is safest, case-sensitive volumes can
                    // be mounted within volumes that
                    // are case-insensitive.
                    if (destinationPath.StartsWith(extractPath,
StringComparison.OrdinalIgnoreCase))
                        entry.ExtractToFile(destinationPath);
                }
            }
        }
    }
}

```

Example 3: Add a file to an existing .zip file

The following example uses the [ZipArchive](#) class to access an existing .zip file, and adds a file to it. The new file gets compressed when you add it to the existing .zip file.

C#

```

using System;
using System.IO;
using System.IO.Compression;

namespace ConsoleApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            using (FileStream zipToOpen = new
FileStream(@"c:\users\exampleuser\release.zip", FileMode.Open))
            {

```

```

        using (ZipArchive archive = new ZipArchive(zipToOpen,
ZipArchiveMode.Update))
        {
            ZipArchiveEntry readmeEntry =
archive.CreateEntry("Readme.txt");
            using (StreamWriter writer = new
StreamWriter(readmeEntry.Open()))
            {
                writer.WriteLine("Information about this
package.");
                writer.WriteLine("=====");
            }
        }
    }
}

```

Example 4: Compress and decompress .gz files

You can also use the [GZipStream](#) and [DeflateStream](#) classes to compress and decompress data. They use the same compression algorithm. You can decompress [GZipStream](#) objects that are written to a .gz file by using many common tools. The following example shows how to compress and decompress a directory of files by using the [GZipStream](#) class:

C#

```

using System;
using System.IO;
using System.IO.Compression;

public class Program
{
    private static string directoryPath = @"..\temp";
    public static void Main()
    {
        DirectoryInfo directorySelected = new DirectoryInfo(directoryPath);
        Compress(directorySelected);

        foreach (FileInfo fileToDecompress in
directorySelected.GetFiles("*.gz"))
        {
            Decompress(fileToDecompress);
        }
    }

    public static void Compress(DirectoryInfo directorySelected)
    {
        foreach (FileInfo fileToCompress in directorySelected.GetFiles())
        {
            fileToCompress.Compress();
        }
    }

    public static void Decompress(FileInfo fileToDecompress)
    {
        fileToDecompress.Decompress();
    }
}

```

```

    {
        using (FileStream originalFileStream =
fileToCompress.OpenRead())
        {
            if ((File.GetAttributes(fileToCompress.FullName) &
                FileAttributes.Hidden) != FileAttributes.Hidden &
fileToCompress.Extension != ".gz")
            {
                using (FileStream compressedFileStream =
File.Create(fileToCompress.FullName + ".gz"))
                {
                    using (GZipStream compressionStream = new
GZipStream(compressedFileStream,
                        CompressionMode.Compress))
                    {
                        originalFileStream.CopyTo(compressionStream);
                    }
                }
                FileInfo info = new FileInfo(directoryPath +
Path.DirectorySeparatorChar + fileToCompress.Name + ".gz");
                Console.WriteLine($"Compressed {fileToCompress.Name}
from {fileToCompress.Length.ToString()} to {info.Length.ToString()}
bytes.");
            }
        }
    }

    public static void Decompress(FileInfo fileToDecompress)
{
    using (FileStream originalFileStream = fileToDecompress.OpenRead())
    {
        string currentFileName = fileToDecompress.FullName;
        string newFileName =
currentFileName.Remove(currentFileName.Length -
fileToDecompress.Extension.Length);

        using (FileStream decompressedFileStream =
File.Create(newFileName))
        {
            using (GZipStream decompressionStream = new
GZipStream(originalFileStream, CompressionMode.Decompress))
            {
                decompressionStream.CopyTo(decompressedFileStream);
                Console.WriteLine($"Decompressed:
{fileToDecompress.Name}");
            }
        }
    }
}
}

```

See also

- [ZipArchive](#)
- [ZipFile](#)
- [ZipArchiveEntry](#)
- [DeflateStream](#)
- [GZipStream](#)
- [File and stream I/O](#)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback [here](#).

 [Open a documentation issue](#)

 [Provide product feedback](#)

Compose streams

Article • 09/15/2021

A *backing store* is a storage medium, such as a disk or memory. Each different backing store implements its own stream as an implementation of the [Stream](#) class.

Each stream type reads and writes bytes from and to its given backing store. Streams that connect to backing stores are called *base streams*. Base streams have constructors with the parameters necessary to connect the stream to the backing store. For example, [FileStream](#) has constructors that specify a path parameter, which specifies how the file will be shared by processes.

The design of the [System.IO](#) classes provides simplified stream composition. You can attach base streams to one or more pass-through streams that provide the functionality you want. You can attach a reader or writer to the end of the chain, so the preferred types can be read or written easily.

The following code examples create a [FileStream](#) around the existing *MyFile.txt* in order to buffer *MyFile.txt*. Note that [FileStreams](#) are buffered by default.

ⓘ Important

The examples assume that a file named *MyFile.txt* already exists in the same folder as the app.

Example: Use StreamReader

The following example creates a [StreamReader](#) to read characters from the [FileStream](#), which is passed to the [StreamReader](#) as its constructor argument.

[StreamReader.ReadLine](#) then reads until [StreamReader.Peek](#) finds no more characters.

C#

```
using System;
using System.IO;

public class CompBuf
{
    private const string FILE_NAME = "MyFile.txt";

    public static void Main()
    {
        if (!File.Exists(FILE_NAME))

```

```
        {
            Console.WriteLine($"{FILE_NAME} does not exist!");
            return;
        }
        // Create an instance of StreamReader characters from the file.
        using (StreamReader sr = new StreamReader(FILE_NAME))
        {
            string input;
            // While not at the end of the file, read lines from the file.
            while (sr.Peek() > -1)
            {
                input = sr.ReadLine();
                Console.WriteLine(input);
            }
        }
    }
}
```

Example: Use BinaryReader

The following example creates a [BinaryReader](#) to read bytes from the [FileStream](#), which is passed to the [BinaryReader](#) as its constructor argument. [ReadByte](#) then reads until [PeekChar](#) finds no more bytes.

C#

```
using System;
using System.IO;

public class ReadBuf
{
    private const string FILE_NAME = "MyFile.txt";

    public static void Main()
    {
        if (!File.Exists(FILE_NAME))
        {
            Console.WriteLine($"{FILE_NAME} does not exist.");
            return;
        }
        using (FileStream f = new FileStream(FILE_NAME, FileMode.Open,
FileAccess.Read, FileShare.Read))
        {
            // Create an instance of BinaryReader that can
            // read bytes from the FileStream.
            using (BinaryReader br = new BinaryReader(f))
            {
                byte input;
                // While not at the end of the file, read lines from the
file.
                while (br.PeekChar() > -1)
```

```
        input = br.ReadByte();
        Console.WriteLine(input);
    }
}
}
```

See also

- [StreamReader](#)
- [StreamReader.ReadLine](#)
- [StreamReader.Peek](#)
- [FileStream](#)
- [BinaryReader](#)
- [BinaryReader.ReadByte](#)
- [BinaryReader.PeekChar](#)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

How to: Convert between .NET Framework and Windows Runtime streams (Windows only)

Article • 03/28/2023

.NET Framework for UWP apps is a subset of the full .NET Framework. Because of security and other requirements for UWP apps, you can't use the full set of .NET Framework APIs to open and read files. For more information, see [.NET for UWP apps overview](#). However, you may want to use .NET Framework APIs for other stream manipulation operations. To manipulate these streams, you can convert between a .NET Framework stream type, such as [MemoryStream](#) or [FileStream](#), and a Windows Runtime stream, such as [IInputStream](#), [IOOutputStream](#), or [IRandomAccessStream](#).

The [System.IO.WindowsRuntimeStreamExtensions](#) class contains methods that make these conversions easy. However, underlying differences between .NET Framework and Windows Runtime streams affect the results of using these methods, as described in the following sections:

Convert from a Windows Runtime to a .NET Framework stream

To convert from a Windows Runtime stream to a .NET Framework stream, use one of the following [System.IO.WindowsRuntimeStreamExtensions](#) methods:

- [WindowsRuntimeStreamExtensionsAsStream](#) converts a random-access stream in the Windows Runtime to a managed stream in .NET for UWP apps.
- [WindowsRuntimeStreamExtensionsAsStreamForWrite](#) converts an output stream in the Windows Runtime to a managed stream in .NET for UWP apps.
- [WindowsRuntimeStreamExtensionsAsStreamForRead](#) converts an input stream in the Windows Runtime to a managed stream in .NET for UWP apps.

The Windows Runtime offers stream types that support reading only, writing only, or reading and writing. These capabilities are maintained when you convert a Windows Runtime stream to a .NET Framework stream. Furthermore, if you convert a Windows Runtime stream to a .NET Framework stream and back, you get the original Windows Runtime instance back.

It's best practice to use the conversion method that matches the capabilities of the Windows Runtime stream you want to convert. However, since [IRandomAccessStream](#) is readable and writeable (it implements both [IOutputStream](#) and [IInputStream](#)), the conversion methods maintain the capabilities of the original stream. For example, using [WindowsRuntimeStreamExtensions.AsStreamForRead](#) to convert an [IRandomAccessStream](#) doesn't limit the converted .NET Framework stream to being readable. It's also writable.

Example: Convert Windows Runtime random-access to .NET Framework stream

To convert from a Windows Runtime random-access stream to a .NET Framework stream, use the [WindowsRuntimeStreamExtensionsAsStream](#) method.

The following code example prompts you to select a file, opens it with Windows Runtime APIs, and then converts it to a .NET Framework stream. It reads the stream and outputs it to a text block. You would typically manipulate the stream with .NET Framework APIs before outputting the results.

C#

```
// Create a file picker.
FileOpenPicker picker = new FileOpenPicker();
picker.SuggestedStartLocation = PickerLocationId.DocumentsLibrary;
picker.ViewMode = PickerViewMode.List;
picker.FileTypeFilter.Add(".txt");

// Show picker, enabling user to pick one file.
StorageFile result = await picker.PickSingleFileAsync();
if (result != null)
{
    try
    {
        // Retrieve the stream. This method returns a
        IRandomAccessStreamWithContentType.
        var stream = await result.OpenReadAsync();

        // Convert the stream to a .NET stream using AsStream, pass to a
        // StreamReader and read the stream.
        using (StreamReader sr = new StreamReader(streamAsStream()))
        {
            TextBlock1.Text = sr.ReadToEnd();
        }
    }
    catch (Exception ex)
    {
        // ...
    }
}
```

```
    }  
}
```

Convert from a .NET Framework to a Windows Runtime stream

To convert from a .NET Framework stream to a Windows Runtime stream, use one of the following [System.IO.WindowsRuntimeStreamExtensions](#) methods:

- [WindowsRuntimeStreamExtensions.AsInputStream](#) converts a managed stream in .NET for UWP apps to an input stream in the Windows Runtime.
- [WindowsRuntimeStreamExtensions.AsOutputStream](#) converts a managed stream in .NET for UWP apps to an output stream in the Windows Runtime.
- [WindowsRuntimeStreamExtensions.AsRandomAccessStream](#) converts a managed stream in .NET for UWP apps to a random-access stream that the Windows Runtime can use for reading or writing.

When you convert a .NET Framework stream to a Windows Runtime stream, the capabilities of the converted stream depend on the original stream. For example, if the original stream supports both reading and writing, and you call [WindowsRuntimeStreamExtensions.AsInputStream](#) to convert the stream, the returned type is an `IRandomAccessStream`. `IRandomAccessStream` implements `IInputStream` and `IOutputStream`, and supports reading and writing.

.NET Framework streams don't support cloning, even after conversion. If you convert a .NET Framework stream to a Windows Runtime stream and call [GetInputStreamAt](#) or [GetOutputStreamAt](#), which call [CloneStream](#), or if you call [CloneStream](#) directly, an exception occurs.

Example: Convert .NET Framework to Windows Runtime random-access stream

To convert from a .NET Framework stream to a Windows Runtime random-access stream, use the [AsRandomAccessStream](#) method, as shown in the following example:

 **Important**

Make sure that the .NET Framework stream you are using supports seeking, or copy it to a stream that does. You can use the `Stream.CanSeek` property to determine this.

C#

```
// Create an HttpClient and access an image as a stream.  
var client = new HttpClient();  
Stream stream = await client.GetStreamAsync("https://learn.microsoft.com/en-us/dotnet/images/hub/featured-1.png");  
// Create a .NET memory stream.  
var memStream = new MemoryStream();  
// Convert the stream to the memory stream, because a memory stream supports seeking.  
await stream.CopyToAsync(memStream);  
// Set the start position.  
memStream.Position = 0;  
// Create a new bitmap image.  
var bitmap = new BitmapImage();  
// Set the bitmap source to the stream, which is converted to a IRandomAccessStream.  
bitmap.SetSource(memStream.AsRandomAccessStream());  
// Set the image control source to the bitmap.  
Image1.Source = bitmap;
```

See also

- [Quickstart: Read and write a file \(Windows\)](#)
- [.NET for Windows Store apps overview](#)
- [.NET for Windows Store apps APIs](#)

 Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback [here](#).

 [Open a documentation issue](#)

 [Provide product feedback](#)

Asynchronous File I/O

Article • 02/13/2023

Asynchronous operations enable you to perform resource-intensive I/O operations without blocking the main thread. This performance consideration is particularly important in a Windows 8.x Store app or desktop app where a time-consuming stream operation can block the UI thread and make your app appear as if it is not working.

Starting with .NET Framework 4.5, the I/O types include `async` methods to simplify asynchronous operations. An `async` method contains `Async` in its name, such as [ReadAsync](#), [WriteAsync](#), [CopyToAsync](#), [FlushAsync](#), [ReadLineAsync](#), and [ReadToEndAsync](#). These `async` methods are implemented on stream classes, such as [Stream](#), [FileStream](#), and [MemoryStream](#), and on classes that are used for reading from or writing to streams, such [TextReader](#) and [TextWriter](#).

In .NET Framework 4 and earlier versions, you have to use methods such as [BeginRead](#) and [EndRead](#) to implement asynchronous I/O operations. These methods are still available in current .NET versions to support legacy code; however, the `async` methods help you implement asynchronous I/O operations more easily.

C# and Visual Basic each have two keywords for asynchronous programming:

- `Async` (Visual Basic) or `async` (C#) modifier, which is used to mark a method that contains an asynchronous operation.
- `Await` (Visual Basic) or `await` (C#) operator, which is applied to the result of an `async` method.

To implement asynchronous I/O operations, use these keywords in conjunction with the `async` methods, as shown in the following examples. For more information, see [Asynchronous programming with async and await \(C#\)](#) or [Asynchronous Programming with Async and Await \(Visual Basic\)](#).

The following example demonstrates how to use two [FileStream](#) objects to copy files asynchronously from one directory to another. Notice that the [Click](#) event handler for the [Button](#) control is marked with the `async` modifier because it calls an asynchronous method.

C#

```
using System;
using System.Threading.Tasks;
using System.Windows;
```

```

using System.IO;

namespace WpfApplication
{
    public partial class MainWindow : Window
    {
        public MainWindow()
        {
            InitializeComponent();
        }

        private async void Button_Click(object sender, RoutedEventArgs e)
        {
            string startDirectory = @"c:\Users\exampleuser\start";
            string endDirectory = @"c:\Users\exampleuser\end";

            foreach (string filename in
Directory.EnumerateFiles(startDirectory))
            {
                using (FileStream sourceStream = File.Open(filename,
 FileMode.Open))
                {
                    using (FileStream destinationStream =
File.Create(Path.Combine(endDirectory, Path.GetFileName(filename))))
                    {
                        await sourceStream.CopyToAsync(destinationStream);
                    }
                }
            }
        }
    }
}

```

The next example is similar to the previous one but uses [StreamReader](#) and [StreamWriter](#) objects to read and write the contents of a text file asynchronously.

C#

```

private async void Button_Click(object sender, RoutedEventArgs e)
{
    string UserDirectory = @"c:\Users\exampleuser\";

    using (StreamReader SourceReader = File.OpenText(UserDirectory +
"BigFile.txt"))
    {
        using (StreamWriter DestinationWriter =
File.CreateText(UserDirectory + "CopiedFile.txt"))
        {
            await CopyFilesAsync(SourceReader, DestinationWriter);
        }
    }
}

```

```
public async Task CopyFilesAsync(StreamReader Source, StreamWriter Destination)
{
    char[] buffer = new char[0x1000];
    int numRead;
    while ((numRead = await Source.ReadAsync(buffer, 0, buffer.Length)) != 0)
    {
        await Destination.WriteAsync(buffer, 0, numRead);
    }
}
```

The next example shows the code-behind file and the XAML file that are used to open a file as a [Stream](#) in a Windows 8.x Store app, and read its contents by using an instance of the [StreamReader](#) class. It uses asynchronous methods to open the file as a stream and to read its contents.

C#

```
using System;
using System.IO;
using System.Text;
using Windows.Storage.Pickers;
using Windows.Storage;
using Windows.UI.Xaml;
using Windows.UI.Xaml.Controls;

namespace ExampleApplication
{
    public sealed partial class BlankPage : Page
    {
        public BlankPage()
        {
            this.InitializeComponent();
        }

        private async void Button_Click_1(object sender, RoutedEventArgs e)
        {
            StringBuilder contents = new StringBuilder();
            string nextLine;
            int lineCounter = 1;

            var openPicker = new FileOpenPicker();
            openPicker.SuggestedStartLocation =
PickerLocationId.DocumentsLibrary;
            openPicker.FileTypeFilter.Add(".txt");
            StorageFile selectedFile = await
openPicker.PickSingleFileAsync();

            using (StreamReader reader = new StreamReader(await
selectedFile.OpenStreamForReadAsync()))
            {
```

```

        while ((nextLine = await reader.ReadLineAsync()) != null)
        {
            contents.AppendFormat("{0}. ", lineCounter);
            contents.Append(nextLine);
            contents.AppendLine();
            lineCounter++;
            if (lineCounter > 3)
            {
                contents.AppendLine("Only first 3 lines shown.");
                break;
            }
        }
        DisplayContentsBlock.Text = contents.ToString();
    }
}

```

XAML

```

<Page
    x:Class="ExampleApplication.BlankPage"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="using:ExampleApplication"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    mc:Ignorable="d">

    <StackPanel Background="{StaticResource ApplicationPageBackgroundBrush}"
    VerticalAlignment="Center" HorizontalAlignment="Center">
        <TextBlock Text="Display lines from a file."></TextBlock>
        <Button Content="Load File" Click="Button_Click_1"></Button>
        <TextBlock Name="DisplayContentsBlock"></TextBlock>
    </StackPanel>
</Page>

```

See also

- [Stream](#)
- [File and Stream I/O](#)
- [Asynchronous programming with async and await \(C#\)](#)
- [Asynchronous Programming with Async and Await \(Visual Basic\)](#)

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

Handling I/O errors in .NET

Article • 09/15/2021

In addition to the exceptions that can be thrown in any method call (such as an [OutOfMemoryException](#) when a system is stressed or an [NullReferenceException](#) due to programmer error), .NET file system methods can throw the following exceptions:

- [System.IO.IOException](#), the base class of all [System.IO](#) exception types. It is thrown for errors whose return codes from the operating system don't directly map to any other exception type.
- [System.IO.FileNotFoundException](#).
- [System.IO.DirectoryNotFoundException](#).
- [DriveNotFoundException](#).
- [System.IO.PathTooLongException](#).
- [System.OperationCanceledException](#).
- [System.UnauthorizedAccessException](#).
- [System.ArgumentException](#), which is thrown for invalid path characters on .NET Framework and on .NET Core 2.0 and previous versions.
- [System.NotSupportedException](#), which is thrown for invalid colons in .NET Framework.
- [System.Security.SecurityException](#), which is thrown for applications running in limited trust that lack the necessary permissions on .NET Framework only. (Full trust is the default on .NET Framework.)

Mapping error codes to exceptions

Because the file system is an operating system resource, I/O methods in both .NET Core and .NET Framework wrap calls to the underlying operating system. When an I/O error occurs in code executed by the operating system, the operating system returns error information to the .NET I/O method. The method then translates the error information, typically in the form of an error code, into a .NET exception type. In most cases, it does this by directly translating the error code into its corresponding exception type; it does not perform any special mapping of the error based on the context of the method call.

For example, on the Windows operating system, a method call that returns an error code of `ERROR_FILE_NOT_FOUND` (or 0x02) maps to a [FileNotFoundException](#), and an error code of `ERROR_PATH_NOT_FOUND` (or 0x03) maps to a [DirectoryNotFoundException](#).

However, the precise conditions under which the operating system returns particular error codes is often undocumented or poorly documented. As a result, unexpected

exceptions can occur. For example, because you are working with a directory rather than a file, you would expect that providing an invalid directory path to the [DirectoryInfo](#) constructor throws a [DirectoryNotFoundException](#). However, it may also throw a [FileNotFoundException](#).

Exception handling in I/O operations

Because of this reliance on the operating system, identical exception conditions (such as the directory not found error in our example) can result in an I/O method throwing any one of the entire class of I/O exceptions. This means that, when calling I/O APIs, your code should be prepared to handle most or all of these exceptions, as shown in the following table:

| Exception type | .NET Core/.NET 5+ | .NET Framework |
|---|---------------------------|--------------------|
| IOException | Yes | Yes |
| FileNotFoundException | Yes | Yes |
| DirectoryNotFoundException | Yes | Yes |
| DriveNotFoundException | Yes | Yes |
| PathTooLongException | Yes | Yes |
| OperationCanceledException | Yes | Yes |
| UnauthorizedAccessException | Yes | Yes |
| ArgumentException | .NET Core 2.0 and earlier | Yes |
| NotSupportedException | No | Yes |
| SecurityException | No | Limited trust only |

Handling IOException

As the base class for exceptions in the [System.IO](#) namespace, [IOException](#) is also thrown for any error code that does not map to a predefined exception type. This means that it can be thrown by any I/O operation.

Important

Because [IOException](#) is the base class of the other exception types in the [System.IO](#) namespace, you should handle in a `catch` block after you've handled the

other I/O-related exceptions.

In addition, starting with .NET Core 2.1, validation checks for path correctness (for example, to ensure that invalid characters are not present in a path) have been removed, and the runtime throws an exception mapped from an operating system error code rather than from its own validation code. The most likely exception to be thrown in this case is an [IOException](#), although any other exception type could also be thrown.

Note that, in your exception handling code, you should always handle the [IOException](#) last. Otherwise, because it is the base class of all other IO exceptions, the catch blocks of derived classes will not be evaluated.

In the case of an [IOException](#), you can get additional error information from the [IOException.HResult](#) property. To convert the HResult value to a Win32 error code, you strip out the upper 16 bits of the 32-bit value. The following table lists error codes that may be wrapped in an [IOException](#).

| HResult | Constant | Description |
|-------------------------|----------|---|
| ERROR_SHARING_VIOLATION | 32 | The file name is missing, or the file or directory is in use. |
| ERROR_FILE_EXISTS | 80 | The file already exists. |
| ERROR_INVALID_PARAMETER | 87 | An argument supplied to the method is invalid. |
| ERROR_ALREADY_EXISTS | 183 | The file or directory already exists. |

You can handle these using a `when` clause in a catch statement, as the following example shows.

C#

```
using System;
using System.IO;
using System.Text;

class Program
{
    static void Main()
    {
        var sw = OpenStream(@".\textfile.txt");
        if (sw is null)
            return;
        sw.WriteLine("This is the first line.");
        sw.WriteLine("This is the second line.");
        sw.Close();
    }
}
```

```
static StreamWriter? OpenStream(string path)
{
    if (path is null)
    {
        Console.WriteLine("You did not supply a file path.");
        return null;
    }

    try
    {
        var fs = new FileStream(path, FileMode.CreateNew);
        return new StreamWriter(fs);
    }
    catch (FileNotFoundException)
    {
        Console.WriteLine("The file or directory cannot be found.");
    }
    catch (DirectoryNotFoundException)
    {
        Console.WriteLine("The file or directory cannot be found.");
    }
    catch (DriveNotFoundException)
    {
        Console.WriteLine("The drive specified in 'path' is invalid.");
    }
    catch (PathTooLongException)
    {
        Console.WriteLine("'path' exceeds the maximum supported path
length.");
    }
    catch (UnauthorizedAccessException)
    {
        Console.WriteLine("You do not have permission to create this
file.");
    }
    catch (IOException e) when ((e.HResult & 0x0000FFFF) == 32)
    {
        Console.WriteLine("There is a sharing violation.");
    }
    catch (IOException e) when ((e.HResult & 0x0000FFFF) == 80)
    {
        Console.WriteLine("The file already exists.");
    }
    catch (IOException e)
    {
        Console.WriteLine($"An exception occurred:\nError code: " +
                      $"{e.HResult & 0x0000FFFF}\nMessage:
{e.Message}");
    }
    return null;
}
```

See also

- [Handling and throwing exceptions in .NET](#)
- [Exception handling \(Task Parallel Library\)](#)
- [Best practices for exceptions](#)
- [How to use specific exceptions in a catch block](#)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

Isolated storage

Article • 10/11/2022

For desktop apps, isolated storage is a data storage mechanism that provides isolation and safety by defining standardized ways of associating code with saved data.

Standardization provides other benefits as well. Administrators can use tools designed to manipulate isolated storage to configure file storage space, set security policies, and delete unused data. With isolated storage, your code no longer needs unique paths to specify safe locations in the file system, and data is protected from other applications that only have isolated storage access. Hard-coded information that indicates where an application's storage area is located is unnecessary.

ⓘ Important

Isolated storage is not available for Windows 8.x Store apps. Instead, use the application data classes in the `Windows.Storage` namespaces included in the Windows Runtime API to store local data and files. For more information, see [Application data](#) in the Windows Dev Center.

Data Compartments and Stores

When an application stores data in a file, the file name and storage location must be carefully chosen to minimize the possibility that the storage location will be known to another application and, therefore, vulnerable to corruption. Without a standard system in place to manage these problems, improvising techniques that minimize storage conflicts can be complex, and the results can be unreliable.

With isolated storage, data is always isolated by user and by assembly. Credentials such as the origin or the strong name of the assembly determine assembly identity. Data can also be isolated by application domain, using similar credentials.

When you use isolated storage, your application saves data to a unique data compartment that is associated with some aspect of the code's identity, such as its publisher or signature. The data compartment is an abstraction, not a specific storage location; it consists of one or more isolated storage files, called stores, which contain the actual directory locations where data is stored. For example, an application might have a data compartment associated with it, and a directory in the file system would implement the store that actually preserves the data for that application. The data saved in the store can be any kind of data, from user preference information to application state. For

the developer, the location of the data compartment is transparent. Stores usually reside on the client, but a server application could use isolated stores to store information by impersonating the user on whose behalf it is functioning. Isolated storage can also store information on a server with a user's roaming profile so that the information will travel with the roaming user.

Quotas for Isolated Storage

A quota is a limit on the amount of isolated storage that can be used. The quota includes bytes of file space as well as the overhead associated with the directory and other information in the store. Isolated storage uses permission quotas, which are storage limits that are set by using [IsolatedStoragePermission](#) objects. If you try to write data that exceeds the quota, an [IsolatedStorageException](#) exception is thrown. Security policy, which can be modified using the .NET Framework Configuration Tool (Mscorcfg.msc), determines which permissions are granted to code. Code that has been granted [IsolatedStoragePermission](#) is restricted to using no more storage than the [UserQuota](#) property allows. However, because code can bypass permission quotas by presenting different user identities, permission quotas serve as guidelines for how code should behave rather than as a firm limit on code behavior.

Quotas are not enforced on roaming stores. Because of this, a slightly higher level of permission is required for code to use them. The enumeration values [AssemblyIsolationByRoamingUser](#) and [DomainIsolationByRoamingUser](#) specify a permission to use isolated storage for a roaming user.

Secure Access

Using isolated storage enables partially trusted applications to store data in a manner that is controlled by the computer's security policy. This is especially useful for downloaded components that a user might want to run cautiously. Security policy rarely grants this kind of code permission when you access the file system by using standard I/O mechanisms. However, by default, code running from the local computer, a local network, or the Internet is granted the right to use isolated storage.

Administrators can limit how much isolated storage an application or a user has available, based on an appropriate trust level. In addition, administrators can remove a user's persisted data completely. To create or access isolated storage, code must be granted the appropriate [IsolatedStorageFilePermission](#) permission.

To access isolated storage, code must have all necessary native platform operating system rights. The access control lists (ACLs) that control which users have the rights to

use the file system must be satisfied. .NET applications already have operating system rights to access isolated storage unless they perform (platform-specific) impersonation. In this case, the application is responsible for ensuring that the impersonated user identity has the proper operating system rights to access isolated storage. This access provides a convenient way for code that is run or downloaded from the web to read and write to a storage area related to a particular user.

To control access to isolated storage, the common language runtime uses [IsolatedStorageFilePermission](#) objects. Each object has properties that specify the following values:

- Allowed usage, which indicates the type of access that is allowed. The values are members of the [IsolatedStorageContainment](#) enumeration. For more information about these values, see the table in the next section.
- Storage quota, as discussed in the preceding section.

The runtime demands [IsolatedStorageFilePermission](#) permission when code first attempts to open a store. It decides whether to grant this permission, based on how much the code is trusted. If the permission is granted, the allowed usage and storage quota values are determined by security policy and by the code's request for [IsolatedStorageFilePermission](#). Security policy is set by using the .NET Framework Configuration Tool (Mscorcfg.msc). All callers in the call stack are checked to ensure that each caller has at least the appropriate allowed usage. The runtime also checks the quota imposed on the code that opened or created the store in which the file is to be saved. If these conditions are satisfied, permission is granted. The quota is checked again every time a file is written to the store.

Application code is not required to request permission because the common language runtime will grant whatever [IsolatedStorageFilePermission](#) is appropriate based on security policy. However, there are good reasons to request specific permissions that your application needs, including [IsolatedStorageFilePermission](#).

Allowed Usage and Security Risks

The allowed usage specified by [IsolatedStorageFilePermission](#) determines the degree to which code will be allowed to create and use isolated storage. The following table shows how the allowed usage specified in the permission corresponds to types of isolation and summarizes the security risks associated with each allowed usage.

| Allowed usage | Isolation types | Security impact |
|---|---|---|
| None | No isolated storage use is allowed. | There is no security impact. |
| DomainIsolationByUser | Isolation by user, domain, and assembly. Each assembly has a separate substore within the domain. Stores that use this permission are also implicitly isolated by computer. | This permission level leaves resources open to unauthorized overuse, although enforced quotas make it more difficult. This is called a denial of service attack. |
| DomainIsolationByRoamingUser | Same as DomainIsolationByUser , but store is saved to a location that will roam if roaming user profiles are enabled and quotas are not enforced. | Because quotas must be disabled, storage resources are more vulnerable to a denial of service attack. |
| AssemblyIsolationByUser | Isolation by user and assembly. Stores that use this permission are also implicitly isolated by computer. | Quotas are enforced at this level to help prevent a denial of service attack. The same assembly in another domain can access this store, opening the possibility that information could be leaked between applications. |
| AssemblyIsolationByRoamingUser | Same as AssemblyIsolationByUser , but store is saved to a location that will roam if roaming user profiles are enabled and quotas are not enforced. | Same as in AssemblyIsolationByUser , but without quotas, the risk of a denial of service attack increases. |
| AdministerIsolatedStorageByUser | Isolation by user. Typically, only administrative or debugging tools use this level of permission. | Access with this permission allows code to view or delete any of a user's isolated storage files or directories (regardless of assembly isolation). Risks include, but are not limited to, leaking information and data loss. |
| UnrestrictedIsolatedStorage | Isolation by all users, domains, and assemblies. Typically, only administrative | This permission creates the potential for a total compromise of all isolated stores for all users. |

| Allowed usage | Isolation types | Security impact |
|---------------|--|-----------------|
| | or debugging tools use this level of permission. | |

Safety of isolated storage components with regard to untrusted data

This section applies to the following frameworks:

- .NET Framework (all versions)
- .NET Core 2.1+
- .NET 5+

.NET Framework and .NET Core offer isolated storage as a mechanism to persist data for a user, an application, or a component. This is a legacy component primarily designed for now-deprecated Code Access Security scenarios.

Various isolated storage APIs and tools can be used to read data across trust boundaries. For example, reading data from a machine-wide scope can aggregate data from other, possibly less-trusted user accounts on the machine. Components or applications that read from machine-wide isolated storage scopes should be aware of the consequences of reading this data.

Security-sensitive APIs that can read from the machine-wide scope

Components or applications that call any of the following APIs read from the machine-wide scope:

- [IsolatedStorageFile.GetEnumerator](#), passing a scope that includes the `IsolatedStorageScope.Machine` flag
- [IsolatedStorageFile.GetMachineStoreForApplication](#)
- [IsolatedStorageFile.GetMachineStoreForAssembly](#)
- [IsolatedStorageFile.GetMachineStoreForDomain](#)
- [IsolatedStorageFile.GetStore](#), passing a scope that includes the `IsolatedStorageScope.Machine` flag
- [IsolatedStorageFile.Remove](#), passing a scope that includes the `IsolatedStorageScope.Machine` flag

The [isolated storage tool](#) `storeadm.exe` is impacted if called with the `/machine` switch, as shown in the following code:

txt

```
storeadm.exe /machine [any-other-switches]
```

The isolated storage tool is provided as part of Visual Studio and the .NET Framework SDK.

If the application doesn't involve calls to the preceding APIs, or if the workflow doesn't involve calling `storeadm.exe` in this manner, this document doesn't apply.

Impact in multi-user environments

As mentioned previously, the security impact from these APIs results from data written from one trust environment is read from a different trust environment. Isolated storage generally uses one of three locations to read and write data:

1. `%LOCALAPPDATA%\IsolatedStorage\`: For example, `C:\Users\<username>\AppData\Local\IsolatedStorage\`, for `User` scope.
2. `%APPDATA%\IsolatedStorage\`: For example, `C:\Users\<username>\AppData\Roaming\IsolatedStorage\`, for `User|Roaming` scope.
3. `%PROGRAMDATA%\IsolatedStorage\`: For example, `C:\ProgramData\IsolatedStorage\`, for `Machine` scope.

The first two locations are isolated per-user. Windows ensures that different user accounts on the same machine cannot access each other's user profile folders. Two different user accounts who use the `User` or `User|Roaming` stores will not see each other's data and cannot interfere with each other's data.

The third location is shared across all user accounts on the machine. Different accounts can read from and write to this location, and they're able to see each other's data.

The preceding paths may differ based on the version of Windows in use.

Now consider a multi-user system with two registered users *Mallory* and *Bob*. Mallory has the ability to access her user profile directory `C:\Users\Mallory\`, and she can access the shared machine-wide storage location `C:\ProgramData\IsolatedStorage\`. She cannot access Bob's user profile directory `C:\Users\Bob\`.

If Mallory wishes to attack Bob, she might write data to the machine-wide storage location, then attempt to influence Bob into reading from the machine-wide store. When Bob runs an app that reads from this store, that app will operate on the data Mallory placed there, but from within the context of Bob's user account. The remainder

of this document contemplates various attack vectors and what steps apps can do to minimize their risk to these attacks.

ⓘ Note

In order for such an attack to take place, Mallory requires:

- A user account on the machine.
- The ability to place a file into a known location on the file system.
- Knowledge that Bob will at some point run an app that attempts to read this data.

These are not threat vectors that apply to standard single-user desktop environments like home PCs or single-employee enterprise workstations.

Elevation of privilege

An **elevation of privilege** attack occurs when Bob's app reads Mallory's file and automatically tries to take some action based on the contents of that payload. Consider an app that reads the contents of a startup script from the machine-wide store and passes those contents to `Process.Start`. If Mallory can place a malicious script inside the machine-wide store, when Bob launches his app:

- His app parses and launches Mallory's malicious script *under the context of Bob's user profile*.
- Mallory gains access to Bob's account on the local machine.

Denial of service

A **denial of service** attack occurs when Bob's app reads Mallory's file and crashes or otherwise stops functioning correctly. Consider again the app mentioned previously, which attempts to parse a startup script from the machine-wide store. If Mallory can place a file with malformed contents inside the machine-wide store, she might:

- Cause Bob's app to throw an exception early in the startup path.
- Prevent the app from launching successfully because of the exception.

She has then denied Bob the ability to launch the app under his own user account.

Information disclosure

An **information disclosure** attack occurs when Mallory can trick Bob into disclosing the contents of a file that Mallory does not normally have access to. Consider that Bob has a secret file `C:\Users\Bob\secret.txt` that Mallory wants to read. She knows the path to this file, but she cannot read it because Windows forbids her from gaining access to Bob's user profile directory.

Instead, Mallory places a hard link into the machine-wide store. This is a special kind of file that itself does not contain any contents, rather, it points to another file on disk.

Attempting to read the hard link file will instead read the contents of the file targeted by the link. After creating the hard link, Mallory still cannot read the file contents because she does not have access to the target (`c:\Users\Bob\secret.txt`) of the link. However, Bob *does* have access to this file.

When Bob's app reads from the machine-wide store, it now inadvertently reads the contents of his `secret.txt` file, just as if the file itself had been present in the machine-wide store. When Bob's app exits, if it attempts to resave the file to the machine-wide store, it will end up placing an actual copy of the file in the `*C:\ProgramData\IsolatedStorage*` directory. Since this directory is readable by any user on the machine, Mallory can now read the contents of the file.

Best practices to defend against these attacks

Important: If your environment has multiple mutually untrusted users, **do not** call the API `IsolatedStorageFile.GetEnumerator(IsolatedStorageScope.Machine)` or invoke the tool `storeadm.exe /machine /list`. Both of these assume that they're operating on trusted data. If an attacker can seed a malicious payload in the machine-wide store, that payload can lead to an elevation of privilege attack under the context of the user who runs these commands.

If operating in a multi-user environment, reconsider use of isolated storage features that target the *Machine* scope. If an app must read data from a machine-wide location, prefer to read the data from a location that's writable only by admin accounts. The `%PROGRAMFILES%` directory and the `HKLM` registry hive are examples of locations that are writable by only administrators and readable by everyone. Data read from those locations is therefore considered trustworthy.

If an app must use the *Machine* scope in a multi-user environment, validate the contents of any file that you read from the machine-wide store. If the app is deserializing object graphs from these files, consider using safer serializers like `XmlSerializer` instead of dangerous serializers like `BinaryFormatter` or `NetDataContractSerializer`. Use caution

with deeply nested object graphs or object graphs that perform resource allocation based on the file contents.

Isolated Storage Locations

Sometimes it is helpful to verify a change to isolated storage by using the file system of the operating system. You might also want to know the location of isolated storage files. This location is different depending on the operating system. The following table shows the root locations where isolated storage is created on a few common operating systems. Look for Microsoft\IsolatedStorage directories under this root location. You must change folder settings to show hidden files and folders in order to see isolated storage in the file system.

| Operating system | Location in file system |
|--|---|
| Windows 2000, Windows XP, Windows Server 2003 (upgrade from Windows NT 4.0) | <p>Roaming-enabled stores = <SYSTEMROOT>\Profiles\<user>\Application Data</p> <p>Nonroaming stores = <SYSTEMROOT>\Profiles\<user>\Local Settings\Application Data</p> |
| Windows 2000 - clean installation (and upgrades from Windows 98 and Windows NT 3.51) | <p>Roaming-enabled stores = <SYSTEMDRIVE>\Documents and Settings\<user>\Application Data</p> <p>Nonroaming stores = <SYSTEMDRIVE>\Documents and Settings\<user>\Local Settings\Application Data</p> |
| Windows XP, Windows Server 2003 - clean installation (and upgrades from Windows 2000 and Windows 98) | <p>Roaming-enabled stores = <SYSTEMDRIVE>\Documents and Settings\<user>\Application Data</p> <p>Nonroaming stores = <SYSTEMDRIVE>\Documents and Settings\<user>\Local Settings\Application Data</p> |
| Windows 8, Windows 7, Windows Server 2008, Windows Vista | Roaming-enabled stores = |

| Operating system | Location in file system |
|------------------|---|
| | <SYSTEMDRIVE>\Users\ <user>\AppData\Roaming |
| | Nonroaming stores = <SYSTEMDRIVE>\Users\<user>\AppData\Local |

Creating, Enumerating, and Deleting Isolated Storage

.NET provides three classes in the [System.IO.IsolatedStorage](#) namespace to help you perform tasks that involve isolated storage:

- [IsolatedStorageFile](#), derives from [System.IO.IsolatedStorage.IsolatedStorage](#) and provides basic management of stored assembly and application files. An instance of the [IsolatedStorageFile](#) class represents a single store located in the file system.
- [IsolatedStorageFileStream](#) derives from [System.IO.FileStream](#) and provides access to the files in a store.
- [IsolatedStorageScope](#) is an enumeration that enables you to create and select a store with the appropriate isolation type.

The isolated storage classes enable you to create, enumerate, and delete isolated storage. The methods for performing these tasks are available through the [IsolatedStorageFile](#) object. Some operations require you to have the [IsolatedStorageFilePermission](#) permission that represents the right to administer isolated storage; you might also need to have operating system rights to access the file or directory.

For a series of examples that demonstrate common isolated storage tasks, see the how-to topics listed in [Related Topics](#).

Scenarios for Isolated Storage

Isolated storage is useful in many situations, including these four scenarios:

- Downloaded controls. Managed code controls downloaded from the Internet are not allowed to write to the hard drive through normal I/O classes, but they can use isolated storage to persist users' settings and application states.

- Shared component storage. Components that are shared between applications can use isolated storage to provide controlled access to data stores.
- Server storage. Server applications can use isolated storage to provide individual stores for a large number of users making requests to the application. Because isolated storage is always segregated by user, the server must impersonate the user making the request. In this case, data is isolated based on the identity of the principal, which is the same identity the application uses to distinguish between its users.
- Roaming. Applications can also use isolated storage with roaming user profiles. This allows a user's isolated stores to roam with the profile.

Do not use isolated storage in the following situations:

- To store high-value secrets, such as unencrypted keys or passwords, because isolated storage is not protected from highly trusted code, from unmanaged code, or from trusted users of the computer.
- To store code.
- To store configuration and deployment settings, which administrators control. (User preferences are not considered to be configuration settings because administrators do not control them.)

Many applications use a database to store and isolate data, in which case one or more rows in a database might represent storage for a specific user. You might choose to use isolated storage instead of a database when the number of users is small, when the overhead of using a database is significant, or when no database facility exists. Also, when the application requires storage that is more flexible and complex than what a row in a database provides, isolated storage can provide a viable alternative.

Related articles

| Title | Description |
|---|---|
| Types of Isolation | Describes the different types of isolation. |
| How to: Obtain Stores for Isolated Storage | Provides an example of using the IsolatedStorageFile class to obtain a store isolated by user and assembly. |
| How to: Enumerate Stores for Isolated Storage | Shows how to use the IsolatedStorageFile.GetEnumerator method to calculate the size of all isolated storage for the user. |

| Title | Description |
|--|--|
| How to: Delete Stores in Isolated Storage | Shows how to use the <code>IsolatedStorageFile.Remove</code> method in two different ways to delete isolated stores. |
| How to: Anticipate Out-of-Space Conditions with Isolated Storage | Shows how to measure the remaining space in an isolated store. |
| How to: Create Files and Directories in Isolated Storage | Provides some examples of creating files and directories in an isolated store. |
| How to: Find Existing Files and Directories in Isolated Storage | Demonstrates how to read the directory structure and files in isolated storage. |
| How to: Read and Write to Files in Isolated Storage | Provides an example of writing a string to an isolated storage file and reading it back. |
| How to: Delete Files and Directories in Isolated Storage | Demonstrates how to delete isolated storage files and directories. |
| File and Stream I/O | Explains how you can perform synchronous and asynchronous file and data stream access. |

Reference

- [System.IO.IsolatedStorage.IsolatedStorage](#)
- [System.IO.IsolatedStorage.IsolatedStorageFile](#)
- [System.IO.IsolatedStorage.IsolatedStorageFileStream](#)
- [System.IO.IsolatedStorage.IsolatedStorageScope](#)

 Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

Types of isolation

Article • 09/15/2021

Access to isolated storage is always restricted to the user who created it. To implement this type of isolation, the common language runtime uses the same notion of user identity that the operating system recognizes, which is the identity associated with the process in which the code is running when the store is opened. This identity is an authenticated user identity, but impersonation can cause the identity of the current user to change dynamically.

Access to isolated storage is also restricted according to the identity associated with the application's domain and assembly, or with the assembly alone. The runtime obtains these identities in the following ways:

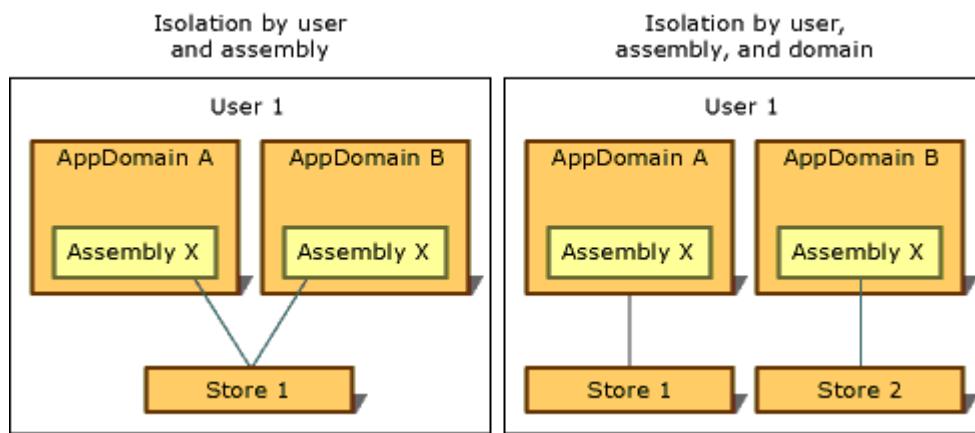
- Domain identity represents the evidence of the application, which in the case of a web application might be the full URL. For shell-hosted code, the domain identity might be based on the application directory path. For example, if the executable runs from the path C:\Office\MyApp.exe, the domain identity would be C:\Office\MyApp.exe.
- Assembly identity is the evidence of the assembly. This might come from a cryptographic digital signature, which can be the assembly's [strong name](#), the software publisher of the assembly, or its URL identity. If an assembly has both a strong name and a software publisher identity, then the software publisher identity is used. If the assembly comes from the Internet and is unsigned, the URL identity is used. For more information about assemblies and strong names, see [Programming with Assemblies](#).
- Roaming stores move with a user that has a roaming user profile. Files are written to a network directory and are downloaded to any computer the user logs into. For more information about roaming user profiles, see [IsolatedStorageScope.Roaming](#).

By combining the concepts of user, domain, and assembly identity, isolated storage can isolate data in the following ways, each of which has its own usage scenarios:

- [Isolation by user and assembly](#)
- [Isolation by user, domain, and assembly](#)

Either of these isolations can be combined with a roaming user profile. For more information, see the section [Isolated Storage and Roaming](#).

The following illustration demonstrates how stores are isolated in different scopes:



Except for roaming stores, isolated storage is always implicitly isolated by computer because it uses the storage facilities that are local to a given computer.

ⓘ Important

Isolated storage is not available for Windows 8.x Store apps. Instead, use the application data classes in the `Windows.Storage` namespaces included in the Windows Runtime API to store local data and files. For more information, see [Application data](#) in the Windows Dev Center.

Isolation by User and Assembly

When the assembly that uses the data store needs to be accessible from any application's domain, isolation by user and assembly is appropriate. Typically, in this situation, isolated storage is used to store data that applies across multiple applications and is not tied to any particular application, such as the user's name or license information. To access storage isolated by user and assembly, code must be trusted to transfer information between applications. Typically, isolation by user and assembly is allowed on intranets but not on the Internet. Calling the static `IsolatedStorageFile.GetStore` method and passing in a user and an assembly `IsolatedStorageScope` returns storage with this kind of isolation.

The following code example retrieves a store that is isolated by user and assembly. The store can be accessed through the `isoFile` object.

C#

```
IsolatedStorageFile isoFile =
    IsolatedStorageFile.GetStore(IsolatedStorageScope.User |
    IsolatedStorageScope.Assembly, null, null);
```

For an example that uses the evidence parameters, see [GetStore\(IsolatedStorageScope, Evidence, Type, Evidence, Type\)](#).

The [GetUserStoreForAssembly](#) method is available as a shortcut, as shown in the following code example. This shortcut cannot be used to open stores that are capable of roaming; use [GetStore](#) in such cases.

```
C#
```

```
IsolatedStorageFile isoFile = IsolatedStorageFile.GetUserStoreForAssembly();
```

Isolation by User, Domain, and Assembly

If your application uses a third-party assembly that requires a private data store, you can use isolated storage to store the private data. Isolation by user, domain, and assembly ensures that only code in a given assembly can access the data, and only when the assembly is used by the application that was running when the assembly created the store, and only when the user for whom the store was created runs the application. Isolation by user, domain, and assembly keeps the third-party assembly from leaking data to other applications. This isolation type should be your default choice if you know that you want to use isolated storage but are not sure which type of isolation to use. Calling the static [GetStore](#) method of [IsolatedStorageFile](#) and passing in a user, domain, and assembly [IsolatedStorageScope](#) returns storage with this kind of isolation.

The following code example retrieves a store isolated by user, domain, and assembly. The store can be accessed through the `isoFile` object.

```
C#
```

```
IsolatedStorageFile isoFile =
    IsolatedStorageFile.GetStore(IsolatedStorageScope.User |
        IsolatedStorageScope.Domain |
        IsolatedStorageScope.Assembly, null, null);
```

Another method is available as a shortcut, as shown in the following code example. This shortcut cannot be used to open stores that are capable of roaming; use [GetStore](#) in such cases.

```
C#
```

```
IsolatedStorageFile isoFile = IsolatedStorageFile.GetUserStoreForDomain();
```

Isolated Storage and Roaming

Roaming user profiles are a Windows feature that enables a user to set up an identity on a network and use that identity to log into any network computer, carrying over all personalized settings. An assembly that uses isolated storage can specify that the user's isolated storage should move with the roaming user profile. Roaming can be used in conjunction with isolation by user and assembly or with isolation by user, domain, and assembly. If a roaming scope is not used, stores will not roam even if a roaming user profile is used.

The following code example retrieves a roaming store isolated by user and assembly. The store can be accessed through the `isoFile` object.

C#

```
IsolatedStorageFile isoFile =
    IsolatedStorageFile.GetStore(IsolatedStorageScope.User |
        IsolatedStorageScope.Assembly |
        IsolatedStorageScope.Roaming, null, null);
```

A domain scope can be added to create a roaming store isolated by user, domain, and application. The following code example demonstrates this.

C#

```
IsolatedStorageFile isoFile =
    IsolatedStorageFile.GetStore(IsolatedStorageScope.User |
        IsolatedStorageScope.Assembly | IsolatedStorageScope.Domain |
        IsolatedStorageScope.Roaming, null, null);
```

See also

- [IsolatedStorageScope](#)
- [Isolated Storage](#)

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

more information, see [our contributor guide](#).

 [Provide product feedback](#)

How to: Obtain Stores for Isolated Storage

Article • 09/15/2021

An isolated store exposes a virtual file system within a data compartment. The [IsolatedStorageFile](#) class supplies a number of methods for interacting with an isolated store. To create and retrieve stores, [IsolatedStorageFile](#) provides three static methods:

- [GetUserStoreForAssembly](#) returns storage that is isolated by user and assembly.
- [GetUserStoreForDomain](#) returns storage that is isolated by domain and assembly.

Both methods retrieve a store that belongs to the code from which they are called.

- The static method [GetStore](#) returns an isolated store that is specified by passing in a combination of scope parameters.

The following code returns a store that is isolated by user, assembly, and domain.

C#

```
IsolatedStorageFile isoStore =  
    IsolatedStorageFile.GetStore(IsolatedStorageScope.User |  
        IsolatedStorageScope.Assembly | IsolatedStorageScope.Domain, null,  
        null);
```

You can use the [GetStore](#) method to specify that a store should roam with a roaming user profile. For details on how to set this up, see [Types of Isolation](#).

Isolated stores obtained from within different assemblies are, by default, different stores. You can access the store of a different assembly or domain by passing in the assembly or domain evidence in the parameters of the [GetStore](#) method. This requires permission to access isolated storage by application domain identity. For more information, see the [GetStore](#) method overloads.

The [GetUserStoreForAssembly](#), [GetUserStoreForDomain](#), and [GetStore](#) methods return an [IsolatedStorageFile](#) object. To help you decide which isolation type is most appropriate for your situation, see [Types of Isolation](#). When you have an isolated storage file object, you can use the isolated storage methods to read, write, create, and delete files and directories.

There is no mechanism that prevents code from passing an [IsolatedStorageFile](#) object to code that does not have sufficient access to get the store itself. Domain and assembly

identities and isolated storage permissions are checked only when a reference to an [IsolatedStorage](#) object is obtained, typically in the [GetUserStoreForAssembly](#), [GetUserStoreForDomain](#), or [GetStore](#) method. Protecting references to [IsolatedStorageFile](#) objects is, therefore, the responsibility of the code that uses these references.

Example

The following code provides a simple example of a class obtaining a store that is isolated by user and assembly. The code can be changed to retrieve a store that is isolated by user, domain, and assembly by adding [IsolatedStorageScope.Domain](#) to the arguments that the [GetStore](#) method passes.

After you run the code, you can confirm that a store was created by typing [StoreAdm /LIST](#) at the command line. This runs the [Isolated Storage tool \(Storeadm.exe\)](#) and lists all the current isolated stores for the user.

C#

```
using System;
using System.IO.IsolatedStorage;

public class ObtainingAStore
{
    public static void Main()
    {
        // Get a new isolated store for this assembly and put it into an
        // isolated store object.

        IsolatedStorageFile isoStore =
IsolatedStorageFile.GetStore(IsolatedStorageScope.User |
    IsolatedStorageScope.Assembly, null, null);
    }
}
```

See also

- [IsolatedStorageFile](#)
- [IsolatedStorageScope](#)
- [Isolated Storage](#)
- [Types of Isolation](#)
- [Assemblies in .NET](#)

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

How to: Enumerate Stores for Isolated Storage

Article • 09/15/2021

You can enumerate all isolated stores for the current user by using the [IsolatedStorageFile.GetEnumerator](#) static method. This method takes an [IsolatedStorageScope](#) value and returns an [IsolatedStorageFile](#) enumerator. To enumerate stores, you must have the [IsolatedStorageFilePermission](#) permission that specifies the [AdministerIsolatedStorageByUser](#) value. If you call the [GetEnumerator](#) method with the [User](#) value, it returns an array of [IsolatedStorageFile](#) objects that are defined for the current user.

Example

The following code example obtains a store that is isolated by user and assembly, creates a few files, and retrieves those files by using the [GetEnumerator](#) method.

C#

```
using System;
using System.IO;
using System.IO.IsolatedStorage;
using System.Collections;

public class EnumeratingStores
{
    public static void Main()
    {
        using (IsolatedStorageFile isoStore =
IsolatedStorageFile.GetStore(IsolatedStorageScope.User |
IsolatedStorageScope.Assembly, null, null))
        {
            isoStore.CreateFile("TestFileA.Txt");
            isoStore.CreateFile("TestFileB.Txt");
            isoStore.CreateFile("TestFileC.Txt");
            isoStore.CreateFile("TestFileD.Txt");
        }

        IEnumerator allFiles =
IsolatedStorageFile.GetEnumerator(IsolatedStorageScope.User);
        long totalsize = 0;

        while (allFiles.MoveNext())
        {
            IsolatedStorageFile storeFile =
(IsolatedStorageFile)allFiles.Current;
```

```
        totalsize += (long)storeFile.UsedSize;  
    }  
  
    Console.WriteLine("The total size = " + totalsize);  
}  
}
```

See also

- [IsolatedStorageFile](#)
- [Isolated Storage](#)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

How to: Delete Stores in Isolated Storage

Article • 09/15/2021

The [IsolatedStorageFile](#) class supplies two methods for deleting isolated storage files:

- The instance method [Remove\(\)](#) does not take any arguments and deletes the store that calls it. No permissions are required for this operation. Any code that can access the store can delete any or all the data inside it.
- The static method [Remove\(IsolatedStorageScope\)](#) takes the [User](#) enumeration value, and deletes all the stores for the user who is running the code. This operation requires [IsolatedStorageFilePermission](#) permission for the [AdministerIsolatedStorageByUser](#) value.

Example

The following code example demonstrates the use of the static and instance [Remove](#) methods. The class obtains two stores; one is isolated for user and assembly and the other is isolated for user, domain, and assembly. The user, domain, and assembly store is then deleted by calling the [Remove\(\)](#) method of the isolated storage file `isoStore1`. Then, all remaining stores for the user are deleted by calling the static method [Remove\(IsolatedStorageScope\)](#).

C#

```
using System;
using System.IO.IsolatedStorage;

public class DeletingStores
{
    public static void Main()
    {
        // Get a new isolated store for this user, domain, and assembly.
        // Put the store into an IsolatedStorageFile object.

        IsolatedStorageFile isoStore1 =
IsolatedStorageFile.GetStore(IsolatedStorageScope.User |
    IsolatedStorageScope.Domain | IsolatedStorageScope.Assembly,
null, null);
        Console.WriteLine("A store isolated by user, assembly, and domain
has been obtained.");

        // Get a new isolated store for user and assembly.
```

```
// Put that store into a different IsolatedStorageFile object.

IsolatedStorageFile isoStore2 =
IsolatedStorageFile.GetStore(IsolatedStorageScope.User |
    IsolatedStorageScope.Assembly, null, null);
Console.WriteLine("A store isolated by user and assembly has been
obtained.");

// The Remove method deletes a specific store, in this case the
// isoStore1 file.

isoStore1.Remove();
Console.WriteLine("The user, domain, and assembly isolated store has
been deleted.");

// This static method deletes all the isolated stores for this user.

IsolatedStorageFile.Remove(IsolatedStorageScope.User);
Console.WriteLine("All isolated stores for this user have been
deleted.");
} // End of Main.
}
```

See also

- [IsolatedStorageFile](#)
- [Isolated Storage](#)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

How to: Anticipate Out-of-Space Conditions with Isolated Storage

Article • 09/15/2021

Code that uses isolated storage is constrained by a [quota](#) that specifies the maximum size for the data compartment in which isolated storage files and directories exist. The quota is defined by security policy and is configurable by administrators. If the maximum allowed size is exceeded when you try to write data, an [IsolatedStorageException](#) exception is thrown and the operation fails. This helps prevent malicious denial-of-service attacks that could cause the application to refuse requests because data storage is filled.

To help you determine whether a given write attempt is likely to fail for this reason, the [IsolatedStorage](#) class provides three read-only properties: [AvailableFreeSpace](#), [UsedSize](#), and [Quota](#). You can use these properties to determine whether writing to the store will cause the maximum allowed size of the store to be exceeded. Keep in mind that isolated storage can be accessed concurrently; therefore, when you compute the amount of remaining storage, the storage space could be consumed by the time you try to write to the store. However, you can use the maximum size of the store to help determine whether the upper limit on available storage is about to be reached.

The [Quota](#) property depends on evidence from the assembly to work properly. For this reason, you should retrieve this property only on [IsolatedStorageFile](#) objects that were created by using the [GetUserStoreForAssembly](#), [GetUserStoreForDomain](#), or [GetStore](#) method. [IsolatedStorageFile](#) objects that were created in any other way (for example, objects that were returned from the [GetEnumerator](#) method) will not return an accurate maximum size.

Example

The following code example obtains an isolated store, creates a few files, and retrieves the [AvailableFreeSpace](#) property. The remaining space is reported in bytes.

C#

```
using System;
using System.IO;
using System.IO.IsolatedStorage;

public class CheckingSpace
{
```

```
public static void Main()
{
    // Get an isolated store for this assembly and put it into an
    // IsolatedStorageFile object.
    IsolatedStorageFile isoStore =
IsolatedStorageFile.GetStore(IsolatedStorageScope.User |
    IsolatedStorageScope.Assembly, null, null);

    // Create a few placeholder files in the isolated store.
    new IsolatedStorageFileStream("InTheRoot.txt", FileMode.Create,
isoStore);
    new IsolatedStorageFileStream("Another.txt", FileMode.Create,
isoStore);
    new IsolatedStorageFileStream("AThird.txt", FileMode.Create,
isoStore);
    new IsolatedStorageFileStream("AFourth.txt", FileMode.Create,
isoStore);
    new IsolatedStorageFileStream("AFifth.txt", FileMode.Create,
isoStore);

    Console.WriteLine(isoStore.AvailableFreeSpace + " bytes of space
remain in this isolated store.");
} // End of Main.
}
```

See also

- [IsolatedStorageFile](#)
- [Isolated Storage](#)
- [How to: Obtain Stores for Isolated Storage](#)

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

How to: Create Files and Directories in Isolated Storage

Article • 09/15/2021

After you've obtained an isolated store, you can create directories and files for storing data. Within a store, file and directory names are specified with respect to the root of the virtual file system.

To create a directory, use the [IsolatedStorageFile.CreateDirectory](#) instance method. If you specify a subdirectory of a directory that doesn't exist, both directories are created. If you specify a directory that already exists, the method returns without creating a directory, and no exception is thrown. However, if you specify a directory name that contains invalid characters, an [IsolatedStorageException](#) exception is thrown.

To create a file, use the [IsolatedStorageFile.CreateFile](#) method.

In the Windows operating system, isolated storage file and directory names are case-insensitive. That is, if you create a file named `ThisFile.txt`, and then create another file named `THISFILE.TXT`, only one file is created. The file name keeps its original casing for display purposes.

Isolated storage file creation will throw an [IsolatedStorageException](#) if the path contains a directory that does not exist.

Example

The following code example illustrates how to create files and directories in an isolated store.

C#

```
using System;
using System.IO;
using System.IO.IsolatedStorage;

public class CreatingFilesDirectories
{
    public static void Main()
    {
        using (IsolatedStorageFile isoStore =
IsolatedStorageFile.GetStore(IsolatedStorageScope.User |
IsolatedStorageScope.Domain | IsolatedStorageScope.Assembly, null, null))
        {
            isoStore.CreateDirectory("TopLevelDirectory");
```

```
isoStore.CreateDirectory("TopLevelDirectory/SecondLevel");

isoStore.CreateDirectory("AnotherTopLevelDirectory/InsideDirectory");
Console.WriteLine("Created directories.");

isoStore.CreateFile("InTheRoot.txt");
Console.WriteLine("Created a new file in the root.");

isoStore.CreateFile("AnotherTopLevelDirectory/InsideDirectory/HereIAm.txt");
Console.WriteLine("Created a new file in the InsideDirectory.");
}

}
```

See also

- [IsolatedStorageFile](#)
- [IsolatedStorageFileStream](#)
- [Isolated Storage](#)

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

How to: Find Existing Files and Directories in Isolated Storage

Article • 09/15/2021

To search for a directory in isolated storage, use the [IsolatedStorageFile.GetDirectoryNames](#) method. This method takes a string that represents a search pattern. You can use both single-character (?) and multi-character (*) wildcard characters in the search pattern, but the wildcard characters must appear in the final portion of the name. For example, `directory1/*ect*` is a valid search string, but `*ect*/directory2` is not.

To search for a file, use the [IsolatedStorageFile.GetFileNames](#) method. The restriction for wildcard characters in search strings that applies to [GetDirectoryNames](#) also applies to [GetFileNames](#).

Neither of these methods is recursive; the [IsolatedStorageFile](#) class does not supply any methods for listing all directories or files in your store. However, recursive methods are shown in the following code example.

Example

The following code example illustrates how to create files and directories in an isolated store. First, a store that is isolated for user, domain, and assembly is retrieved and placed in the `isoStore` variable. The [CreateDirectory](#) method is used to set up a few different directories, and the [IsolatedStorageFileStream\(String, FileMode, IsolatedStorageFile\)](#) constructor creates some files in these directories. The code then loops through the results of the [GetAllDirectories](#) method. This method uses [GetDirectoryName](#) to find all the directory names in the current directory. These names are stored in an array, and then [GetAllDirectories](#) calls itself, passing in each directory it has found. As a result, all the directory names are returned in an array. Next, the code calls the [GetAllFiles](#) method. This method calls [GetAllDirectories](#) to find out the names of all the directories, and then it checks each directory for files by using the [GetFileNames](#) method. The result is returned in an array for display.

C#

```
using System;
using System.IO;
using System.IO.IsolatedStorage;
using System.Collections;
```

```

using System.Collections.Generic;

public class FindingExistingFilesAndDirectories
{
    // Retrieves an array of all directories in the store, and
    // displays the results.
    public static void Main()
    {
        // This part of the code sets up a few directories and files in the
        // store.
        IsolatedStorageFile isoStore =
IsolatedStorageFile.GetStore(IsolatedStorageScope.User |
    IsolatedStorageScope.Assembly, null, null);
        isoStore.CreateDirectory("TopLevelDirectory");
        isoStore.CreateDirectory("TopLevelDirectory/SecondLevel");

        isoStore.CreateDirectory("AnotherTopLevelDirectory/InsideDirectory");
        isoStore.CreateFile("InTheRoot.txt");

        isoStore.CreateFile("AnotherTopLevelDirectory/InsideDirectory/HereIAm.txt");
        // End of setup.

        Console.WriteLine('\r');
        Console.WriteLine("Here is a list of all directories in this
isolated store:");

        foreach (string directory in GetAllDirectories("*", isoStore))
        {
            Console.WriteLine(directory);
        }
        Console.WriteLine('\r');

        // Retrieve all the files in the directory by calling the GetFiles
        // method.

        Console.WriteLine("Here is a list of all the files in this isolated
store:");
        foreach (string file in GetAllFiles("*", isoStore)){
            Console.WriteLine(file);
        }
    } // End of Main.

    // Method to retrieve all directories, recursively, within a store.
    public static List<String> GetAllDirectories(string pattern,
IsolatedStorageFile storeFile)
    {
        // Get the root of the search string.
        string root = Path.GetDirectoryName(pattern);

        if (root != "")
        {
            root += "/";
        }

        // Retrieve directories.
    }
}

```

```

        List<String> directoryList = new List<String>
(storeFile.GetDirectoryNames(pattern));

        // Retrieve subdirectories of matches.
        for (int i = 0, max = directoryList.Count; i < max; i++)
        {
            string directory = directoryList[i] + "/";
            List<String> more = GetAllDirectories(root + directory + "*",
storeFile);

            // For each subdirectory found, add in the base path.
            for (int j = 0; j < more.Count; j++)
            {
                more[j] = directory + more[j];
            }

            // Insert the subdirectories into the list and
            // update the counter and upper bound.
            directoryList.InsertRange(i + 1, more);
            i += more.Count;
            max += more.Count;
        }

        return directoryList;
    }

    public static List<String> GetAllFiles(string pattern,
IsolatedStorageFile storeFile)
{
    // Get the root and file portions of the search string.
    string fileString = Path.GetFileName(pattern);

    List<String> fileList = new List<String>
(storeFile.GetFileNames(pattern));

    // Loop through the subdirectories, collect matches,
    // and make separators consistent.
    foreach (string directory in GetAllDirectories("*", storeFile))
    {
        foreach (string file in storeFile.GetFileNames(directory + "/" +
fileString))
        {
            fileList.Add((directory + "/" + file));
        }
    }

    return fileList;
} // End of GetFiles.
}

```

See also

- [IsolatedStorageFile](#)
- [Isolated Storage](#)

 **Collaborate with us on GitHub**

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

How to: Read and Write to Files in Isolated Storage

Article • 09/15/2021

To read from, or write to, a file in an isolated store, use an [IsolatedStorageFileStream](#) object with a stream reader ([StreamReader](#) object) or stream writer ([StreamWriter](#) object).

Example

The following code example obtains an isolated store and checks whether a file named TestStore.txt exists in the store. If it doesn't exist, it creates the file and writes "Hello Isolated Storage" to the file. If TestStore.txt already exists, the example code reads from the file.

C#

```
using System;
using System.IO;
using System.IO.IsolatedStorage;

namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
        {
            IsolatedStorageFile isoStore =
IsolatedStorageFile.GetStore(IsolatedStorageScope.User |
IsolatedStorageScope.Assembly, null, null);

            if (isoStore.FileExists("TestStore.txt"))
            {
                Console.WriteLine("The file already exists!");
                using (IsolatedStorageFileStream isoStream = new
IsolatedStorageFileStream("TestStore.txt", FileMode.Open, isoStore))
                {
                    using (StreamReader reader = new
StreamReader(isoStream))
                    {
                        Console.WriteLine("Reading contents:");
                        Console.WriteLine(reader.ReadToEnd());
                    }
                }
            }
            else
            {
```

```
        using (IsolatedStorageFileStream isoStream = new
IsolatedStorageFileStream("TestStore.txt", FileMode.CreateNew, isoStore))
    {
        using (StreamWriter writer = new
StreamWriter(isoStream))
    {
        writer.WriteLine("Hello Isolated Storage");
        Console.WriteLine("You have written to the file.");
    }
}
}
```

See also

- [IsolatedStorageFile](#)
- [IsolatedStorageFileStream](#)
- [System.IO.FileMode](#)
- [System.IO.FileAccess](#)
- [System.IO.StreamReader](#)
- [System.IO.StreamWriter](#)
- [File and Stream I/O](#)
- [Isolated Storage](#)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

How to: Delete Files and Directories in Isolated Storage

Article • 09/15/2021

You can delete directories and files within an isolated storage file. Within a store, file and directory names are operating-system dependent and are specified as relative to the root of the virtual file system. They are not case-sensitive on Windows operating systems.

The [System.IO.IsolatedStorage.IsolatedStorageFile](#) class supplies two methods for deleting directories and files: [DeleteDirectory](#) and [DeleteFile](#). An [IsolatedStorageException](#) exception is thrown if you try to delete a file or directory that does not exist. If you include a wildcard character in the name, [DeleteDirectory](#) throws an [IsolatedStorageException](#) exception, and [DeleteFile](#) throws an [ArgumentException](#) exception.

The [DeleteDirectory](#) method fails if the directory contains any files or subdirectories. You can use the [GetFileNames](#) and [GetDirectoryName](#)s methods to retrieve the existing files and directories. For more information about searching the virtual file system of a store, see [How to: Find Existing Files and Directories in Isolated Storage](#).

Example

The following code example creates and then deletes several directories and files.

C#

```
using System;
using System.IO.IsolatedStorage;
using System.IO;

public class DeletingFilesDirectories
{
    public static void Main()
    {
        // Get a new isolated store for this user domain and assembly.
        // Put the store into an isolatedStorageFile object.

        IsolatedStorageFile isoStore =
IsolatedStorageFile.GetStore(IsolatedStorageScope.User |
                           IsolatedStorageScope.Domain | IsolatedStorageScope.Assembly,
null, null);

        Console.WriteLine("Creating Directories:");
    }
}
```

```
// This code creates several different directories.

isoStore.CreateDirectory("TopLevelDirectory");
Console.WriteLine("TopLevelDirectory");
isoStore.CreateDirectory("TopLevelDirectory/SecondLevel");
Console.WriteLine("TopLevelDirectory/SecondLevel");

// This code creates two new directories, one inside the other.

isoStore.CreateDirectory("AnotherTopLevelDirectory/InsideDirectory");
Console.WriteLine("AnotherTopLevelDirectory/InsideDirectory");
Console.WriteLine();

// This code creates a few files and places them in the directories.

Console.WriteLine("Creating Files:");

// This file is placed in the root.

IsolatedStorageFileStream isoStream1 = new
IsolatedStorageFileStream("InTheRoot.txt",
    FileMode.Create, isoStore);
Console.WriteLine("InTheRoot.txt");

isoStream1.Close();

// This file is placed in the InsideDirectory.

IsolatedStorageFileStream isoStream2 = new
IsolatedStorageFileStream(
    "AnotherTopLevelDirectory/InsideDirectory/HereIAm.txt",
FileMode.Create, isoStore);

Console.WriteLine("AnotherTopLevelDirectory/InsideDirectory/HereIAm.txt");
Console.WriteLine();

isoStream2.Close();

Console.WriteLine("Deleting File:");

// This code deletes the HereIAm.txt file.

isoStore.DeleteFile("AnotherTopLevelDirectory/InsideDirectory/HereIAm.txt");

Console.WriteLine("AnotherTopLevelDirectory/InsideDirectory/HereIAm.txt");
Console.WriteLine();

Console.WriteLine("Deleting Directory:");

// This code deletes the InsideDirectory.

isoStore.DeleteDirectory("AnotherTopLevelDirectory/InsideDirectory/");
```

```
        Console.WriteLine("AnotherTopLevelDirectory/InsideDirectory/");
        Console.WriteLine();
    } // End of main.
}
```

See also

- [System.IO.IsolatedStorage.IsolatedStorageFile](#)
- [Isolated Storage](#)

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

Pipe Operations in .NET

Article • 09/15/2021

Pipes provide a means for interprocess communication. There are two types of pipes:

- Anonymous pipes.

Anonymous pipes provide interprocess communication on a local computer.

Anonymous pipes require less overhead than named pipes but offer limited services. Anonymous pipes are one-way and cannot be used over a network. They support only a single server instance. Anonymous pipes are useful for communication between threads, or between parent and child processes where the pipe handles can be easily passed to the child process when it is created.

In .NET, you implement anonymous pipes by using the

[AnonymousPipeServerStream](#) and [AnonymousPipeClientStream](#) classes.

See [How to: Use Anonymous Pipes for Local Interprocess Communication](#).

- Named pipes.

Named pipes provide interprocess communication between a pipe server and one or more pipe clients. Named pipes can be one-way or duplex. They support message-based communication and allow multiple clients to connect simultaneously to the server process using the same pipe name. Named pipes also support impersonation, which enables connecting processes to use their own permissions on remote servers.

In .NET, you implement named pipes by using the [NamedPipeServerStream](#) and [NamedPipeClientStream](#) classes.

See [How to: Use Named Pipes for Network Interprocess Communication](#).

See also

- [File and Stream I/O](#)
- [How to: Use Anonymous Pipes for Local Interprocess Communication](#)
- [How to: Use Named Pipes for Network Interprocess Communication](#)

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

How to: Use Anonymous Pipes for Local Interprocess Communication

Article • 12/14/2022

Anonymous pipes provide interprocess communication on a local computer. They offer less functionality than named pipes, but also require less overhead. You can use anonymous pipes to make interprocess communication on a local computer easier. You cannot use anonymous pipes for communication over a network.

To implement anonymous pipes, use the [AnonymousPipeServerStream](#) and [AnonymousPipeClientStream](#) classes.

Example 1

The following example demonstrates a way to send a string from a parent process to a child process using anonymous pipes. This example creates an [AnonymousPipeServerStream](#) object in a parent process with a [PipeDirection](#) value of [Out](#). The parent process then creates a child process by using a client handle to create an [AnonymousPipeClientStream](#) object. The child process has a [PipeDirection](#) value of [In](#).

The parent process then sends a user-supplied string to the child process. The string is displayed to the console in the child process.

The following example shows the server process.

C#

```
using System;
using System.IO;
using System.IO.Pipes;
using System.Diagnostics;

class PipeServer
{
    static void Main()
    {
        Process pipeClient = new Process();

        pipeClient.StartInfo.FileName = "pipeClient.exe";

        using (AnonymousPipeServerStream pipeServer =
            new AnonymousPipeServerStream(PipeDirection.Out,
                HandleInheritance.Inheritable))
```

```

    {
        Console.WriteLine("[SERVER] Current TransmissionMode: {0}.",
            pipeServer.TransmissionMode);

        // Pass the client process a handle to the server.
        pipeClient.StartInfo.Arguments =
            pipeServer.GetClientHandleAsString();
        pipeClient.StartInfo.UseShellExecute = false;
        pipeClient.Start();

        pipeServer.DisposeLocalCopyOfClientHandle();

        try
        {
            // Read user input and send that to the client process.
            using (StreamWriter sw = new StreamWriter(pipeServer))
            {
                sw.AutoFlush = true;
                // Send a 'sync message' and wait for client to receive
                it.
                sw.WriteLine("SYNC");
                pipeServer.WaitForPipeDrain();
                // Send the console input to the client process.
                Console.Write("[SERVER] Enter text: ");
                sw.WriteLine(Console.ReadLine());
            }
        }
        // Catch the IOException that is raised if the pipe is broken
        // or disconnected.
        catch (IOException e)
        {
            Console.WriteLine("[SERVER] Error: {0}", e.Message);
        }
    }

    pipeClient.WaitForExit();
    pipeClient.Close();
    Console.WriteLine("[SERVER] Client quit. Server terminating.");
}
}

```

Example 2

The following example shows the client process. The server process starts the client process and gives that process a client handle. The resulting executable from the client code should be named `pipeClient.exe` and be copied to the same directory as the server executable before running the server process.

```

using System;
using System.IO;
using System.IO.Pipes;

class PipeClient
{
    static void Main(string[] args)
    {
        if (args.Length > 0)
        {
            using (PipeStream pipeClient =
                new AnonymousPipeClientStream(PipeDirection.In, args[0]))
            {
                Console.WriteLine("[CLIENT] Current TransmissionMode: {0}.",
                    pipeClient.TransmissionMode);

                using (StreamReader sr = new StreamReader(pipeClient))
                {
                    // Display the read text to the console
                    string temp;

                    // Wait for 'sync message' from the server.
                    do
                    {
                        Console.WriteLine("[CLIENT] Wait for sync...");  

                        temp = sr.ReadLine();
                    }
                    while (!temp.StartsWith("SYNC"));

                    // Read the server data and echo to the console.
                    while ((temp = sr.ReadLine()) != null)
                    {
                        Console.WriteLine("[CLIENT] Echo: " + temp);
                    }
                }
            }
            Console.Write("[CLIENT] Press Enter to continue...");  

            Console.ReadLine();
        }
    }
}

```

See also

- [Pipes](#)
- [How to: Use Named Pipes for Network Interprocess Communication](#)

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

How to: Use Named Pipes for Network Interprocess Communication

Article • 12/30/2023

Named pipes provide interprocess communication between a pipe server and one or more pipe clients. They offer more functionality than anonymous pipes, which provide interprocess communication on a local computer. Named pipes support full duplex communication over a network and multiple server instances, message-based communication, and client impersonation, which enables connecting processes to use their own set of permissions on remote servers.

ⓘ Important

.NET on Linux uses Unix Domain Sockets (UDS) for the implementation of these APIs.

To implement name pipes, use the [NamedPipeServerStream](#) and [NamedPipeClientStream](#) classes.

Example 1

The following example demonstrates how to create a named pipe by using the [NamedPipeServerStream](#) class. In this example, the server process creates four threads. Each thread can accept a client connection. The connected client process then supplies the server with a file name. If the client has sufficient permissions, the server process opens the file and sends its contents back to the client.

C#

```
using System;
using System.IO;
using System.IO.Pipes;
using System.Text;
using System.Threading;

public class PipeServer
{
    private static int numThreads = 4;

    public static void Main()
    {
        int i;
```

```

Thread?[] servers = new Thread[numThreads];

Console.WriteLine("\n*** Named pipe server stream with impersonation
example ***\n");
Console.WriteLine("Waiting for client connect...\n");
for (i = 0; i < numThreads; i++)
{
    servers[i] = new Thread(ServerThread);
    servers[i]?.Start();
}
Thread.Sleep(250);
while (i > 0)
{
    for (int j = 0; j < numThreads; j++)
    {
        if (servers[j] != null)
        {
            if (servers[j]!.Join(250))
            {
                Console.WriteLine("Server thread[{0}] finished.",
servers[j]!.ManagedThreadId);
                servers[j] = null;
                i--; // decrement the thread watch count
            }
        }
    }
}
Console.WriteLine("\nServer threads exhausted, exiting.");
}

private static void ServerThread(object? data)
{
    NamedPipeServerStream pipeServer =
        new NamedPipeServerStream("testpipe", PipeDirection.InOut,
numThreads);

    int threadId = Thread.CurrentThread.ManagedThreadId;

    // Wait for a client to connect
    pipeServer.WaitForConnection();

    Console.WriteLine("Client connected on thread[{0}].", threadId);
    try
    {
        // Read the request from the client. Once the client has
        // written to the pipe its security token will be available.

        StreamString ss = new StreamString(pipeServer);

        // Verify our identity to the connected client using a
        // string that the client anticipates.

        ss.WriteString("I am the one true server!");
        string filename = ss.ReadString();
    }
}

```

```

        // Read in the contents of the file while impersonating the
client.

        ReadFileToStream fileReader = new ReadFileToStream(ss,
filename);

        // Display the name of the user we are impersonating.
Console.WriteLine("Reading file: {0} on thread[{1}] as user:
{2}.",
                filename, threadId, pipeServer.GetImpersonationUserName());
pipeServer.RunAsClient(fileReader.Start);
}

// Catch the IOException that is raised if the pipe is broken
// or disconnected.
catch (IOException e)
{
    Console.WriteLine("ERROR: {0}", e.Message);
}
pipeServer.Close();
}

}

// Defines the data protocol for reading and writing strings on our stream
public class StreamString
{
    private Stream ioStream;
    private UnicodeEncoding streamEncoding;

    public StreamString(Stream ioStream)
    {
        this.ioStream = ioStream;
        streamEncoding = new UnicodeEncoding();
    }

    public string ReadString()
    {
        int len = 0;

        len = ioStream.ReadByte() * 256;
        len += ioStream.ReadByte();
        byte[] inBuffer = new byte[len];
        ioStream.Read(inBuffer, 0, len);

        return streamEncoding.GetString(inBuffer);
    }

    public int WriteString(string outString)
    {
        byte[] outBuffer = streamEncoding.GetBytes(outString);
        int len = outBuffer.Length;
        if (len > UInt16.MaxValue)
        {
            len = (int)UInt16.MaxValue;
        }
        ioStream.WriteByte((byte)(len / 256));
        ioStream.WriteByte((byte)(len & 255));
    }
}
```

```

        ioStream.Write(outBuffer, 0, len);
        ioStream.Flush();

    return outBuffer.Length + 2;
}

// Contains the method executed in the context of the impersonated user
public class ReadFileToStream
{
    private string fn;
    private StreamString ss;

    public ReadFileToStream(StreamString str, string filename)
    {
        fn = filename;
        ss = str;
    }

    public void Start()
    {
        string contents = File.ReadAllText(fn);
        ss.WriteString(contents);
    }
}

```

Example 2

The following example shows the client process, which uses the [NamedPipeClientStream](#) class. The client connects to the server process and sends a file name to the server. The example uses impersonation, so the identity that is running the client application must have permission to access the file. The server then sends the contents of the file back to the client. The file contents are then displayed to the console.

```

C#

using System;
using System.Diagnostics;
using System.IO;
using System.IO.Pipes;
using System.Security.Principal;
using System.Text;
using System.Threading;

public class PipeClient
{
    private static int numClients = 4;

    public static void Main(string[] args)
    {

```

```

    if (args.Length > 0)
    {
        if (args[0] == "spawnclient")
        {
            var pipeClient =
                new NamedPipeClientStream(".", "testpipe",
                    PipeDirection.InOut, PipeOptions.None,
                    TokenImpersonationLevel.Impersonation);

            Console.WriteLine("Connecting to server...\n");
            pipeClient.Connect();

            var ss = new StreamString(pipeClient);
            // Validate the server's signature string.
            if (ss.ReadString() == "I am the one true server!")
            {
                // The client security token is sent with the first
write.
                // Send the name of the file whose contents are returned
                // by the server.
                ss.WriteString("c:\\textfile.txt");

                // Print the file to the screen.
                Console.Write(ss.ReadString());
            }
            else
            {
                Console.WriteLine("Server could not be verified.");
            }
            pipeClient.Close();
            // Give the client process some time to display results
before exiting.
            Thread.Sleep(4000);
        }
    }
    else
    {
        Console.WriteLine("\n*** Named pipe client stream with
impersonation example ***\n");
        StartClients();
    }
}

// Helper function to create pipe client processes
private static void StartClients()
{
    string currentProcessName = Environment.CommandLine;

    // Remove extra characters when launched from Visual Studio
    currentProcessName = currentProcessName.Trim("'", ' ');

    currentProcessName = Path.ChangeExtension(currentProcessName,
".exe");
    Process?[] plist = new Process?[numClients];
}

```

```

        Console.WriteLine("Spawning client processes...\n");

        if (currentProcessName.Contains(Environment.CurrentDirectory))
        {
            currentProcessName =
currentProcessName.Replace(Environment.CurrentDirectory, String.Empty);
        }

        // Remove extra characters when launched from Visual Studio
        currentProcessName = currentProcessName.Replace("\\\\", String.Empty);
        currentProcessName = currentProcessName.Replace("\\\"", String.Empty);

        int i;
        for (i = 0; i < numClients; i++)
        {
            // Start 'this' program but spawn a named pipe client.
            plist[i] = Process.Start(currentProcessName, "spawnclient");
        }
        while (i > 0)
        {
            for (int j = 0; j < numClients; j++)
            {
                if (plist[j] != null)
                {
                    if (plist[j]!.HasExited)
                    {
                        Console.WriteLine($"Client process[{plist[j]? .Id}] has exited.");
                        plist[j] = null;
                        i--; // decrement the process watch count
                    }
                    else
                    {
                        Thread.Sleep(250);
                    }
                }
            }
        }
        Console.WriteLine("\nClient processes finished, exiting.");
    }
}

// Defines the data protocol for reading and writing strings on our stream.
public class StreamString
{
    private Stream ioStream;
    private UnicodeEncoding streamEncoding;

    public StreamString(Stream ioStream)
    {
        this.ioStream = ioStream;
        streamEncoding = new UnicodeEncoding();
    }

    public string ReadString()

```

```

{
    int len;
    len = ioStream.ReadByte() * 256;
    len += ioStream.ReadByte();
    var inBuffer = new byte[len];
    ioStream.Read(inBuffer, 0, len);

    return streamEncoding.GetString(inBuffer);
}

public int WriteString(string outString)
{
    byte[] outBuffer = streamEncoding.GetBytes(outString);
    int len = outBuffer.Length;
    if (len > UInt16.MaxValue)
    {
        len = (int)UInt16.MaxValue;
    }
    ioStream.WriteByte((byte)(len / 256));
    ioStream.WriteByte((byte)(len & 255));
    ioStream.Write(outBuffer, 0, len);
    ioStream.Flush();

    return outBuffer.Length + 2;
}
}

```

Robust Programming

The client and server processes in this example are intended to run on the same computer, so the server name provided to the `NamedPipeClientStream` object is `".."`. If the client and server processes were on separate computers, `".."` would be replaced with the network name of the computer that runs the server process.

See also

- [TokenImpersonationLevel](#)
- [GetImpersonationUserName](#)
- [Pipes](#)
- [How to: Use Anonymous Pipes for Local Interprocess Communication](#)

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET is an open source project.
Select a link to provide feedback:

-  [Open a documentation issue](#)
-  [Provide product feedback](#)

System.IO.Pipelines in .NET

Article • 12/14/2022

[System.IO.Pipelines](#) is a library that is designed to make it easier to do high-performance I/O in .NET. It's a library targeting .NET Standard that works on all .NET implementations.

The library is available in the [System.IO.Pipelines](#) Nuget package.

What problem does System.IO.Pipelines solve

Apps that parse streaming data are composed of boilerplate code having many specialized and unusual code flows. The boilerplate and special case code is complex and difficult to maintain.

[System.IO.Pipelines](#) was architected to:

- Have high performance parsing streaming data.
- Reduce code complexity.

The following code is typical for a TCP server that receives line-delimited messages (delimited by `'\n'`) from a client:

C#

```
async Task ProcessLinesAsync(NetworkStream stream)
{
    var buffer = new byte[1024];
    await stream.ReadAsync(buffer, 0, buffer.Length);

    // Process a single line from the buffer
    ProcessLine(buffer);
}
```

The preceding code has several problems:

- The entire message (end of line) might not be received in a single call to `ReadAsync`.
- It's ignoring the result of `stream.ReadAsync`. `stream.ReadAsync` returns how much data was read.
- It doesn't handle the case where multiple lines are read in a single `ReadAsync` call.
- It allocates a `byte` array with each read.

To fix the preceding problems, the following changes are required:

- Buffer the incoming data until a new line is found.
- Parse all the lines returned in the buffer.
- It's possible that the line is bigger than 1 KB (1024 bytes). The code needs to resize the input buffer until the delimiter is found in order to fit the complete line inside the buffer.
 - If the buffer is resized, more buffer copies are made as longer lines appear in the input.
 - To reduce wasted space, compact the buffer used for reading lines.
- Consider using buffer pooling to avoid allocating memory repeatedly.
- The following code addresses some of these problems:

C#

```
async Task ProcessLinesAsync(NetworkStream stream)
{
    byte[] buffer = ArrayPool<byte>.Shared.Rent(1024);
    var bytesBuffered = 0;
    var bytesConsumed = 0;

    while (true)
    {
        // Calculate the amount of bytes remaining in the buffer.
        var bytesRemaining = buffer.Length - bytesBuffered;

        if (bytesRemaining == 0)
        {
            // Double the buffer size and copy the previously buffered data
            // into the new buffer.
            var newBuffer = ArrayPool<byte>.Shared.Rent(buffer.Length * 2);
            Buffer.BlockCopy(buffer, 0, newBuffer, 0, buffer.Length);
            // Return the old buffer to the pool.
            ArrayPool<byte>.Shared.Return(buffer);
            buffer = newBuffer;
            bytesRemaining = buffer.Length - bytesBuffered;
        }

        var bytesRead = await stream.ReadAsync(buffer, bytesBuffered,
bytesRemaining);
        if (bytesRead == 0)
        {
            // EOF
            break;
        }

        // Keep track of the amount of buffered bytes.
    }
}
```

```

        bytesBuffered += bytesRead;
        var linePosition = -1;

        do
        {
            // Look for a EOL in the buffered data.
            linePosition = Array.IndexOf(buffer, (byte)'\n', bytesConsumed,
                bytesBuffered - bytesConsumed);

            if (linePosition >= 0)
            {
                // Calculate the length of the line based on the offset.
                var lineLength = linePosition - bytesConsumed;

                // Process the line.
                ProcessLine(buffer, bytesConsumed, lineLength);

                // Move the bytesConsumed to skip past the line consumed
                // (including \n).
                bytesConsumed += lineLength + 1;
            }
        }
        while (linePosition >= 0);
    }
}

```

The previous code is complex and doesn't address all the problems identified. High-performance networking usually means writing complex code to maximize performance. `System.IO.Pipelines` was designed to make writing this type of code easier.

Pipe

The `Pipe` class can be used to create a `PipeWriter/PipeReader` pair. All data written into the `PipeWriter` is available in the `PipeReader`:

C#

```

var pipe = new Pipe();
PipeReader reader = pipe.Reader;
PipeWriter writer = pipe.Writer;

```

Pipe basic usage

C#

```

async Task ProcessLinesAsync(Socket socket)
{

```

```

        var pipe = new Pipe();
        Task writing = FillPipeAsync(socket, pipe.Writer);
        Task reading = ReadPipeAsync(pipe.Reader);

        await Task.WhenAll(reading, writing);
    }

async Task FillPipeAsync(Socket socket, PipeWriter writer)
{
    const int minimumBufferSize = 512;

    while (true)
    {
        // Allocate at least 512 bytes from the PipeWriter.
        Memory<byte> memory = writer.GetMemory(minimumBufferSize);
        try
        {
            int bytesRead = await socket.ReceiveAsync(memory,
SocketFlags.None);
            if (bytesRead == 0)
            {
                break;
            }
            // Tell the PipeWriter how much was read from the Socket.
            writer.Advance(bytesRead);
        }
        catch (Exception ex)
        {
            LogError(ex);
            break;
        }

        // Make the data available to the PipeReader.
        FlushResult result = await writer.FlushAsync();

        if (result.IsCompleted)
        {
            break;
        }
    }

    // By completing PipeWriter, tell the PipeReader that there's no more
    data coming.
    await writer.CompleteAsync();
}

async Task ReadPipeAsync(PipeReader reader)
{
    while (true)
    {
        ReadResult result = await reader.ReadAsync();
        ReadOnlySequence<byte> buffer = result.Buffer;

        while (TryReadLine(ref buffer, out ReadOnlySequence<byte> line))
        {

```

```

        // Process the line.
        ProcessLine(line);
    }

    // Tell the PipeReader how much of the buffer has been consumed.
    reader.AdvanceTo(buffer.Start, buffer.End);

    // Stop reading if there's no more data coming.
    if (result.IsCompleted)
    {
        break;
    }
}

// Mark the PipeReader as complete.
await reader.CompleteAsync();
}

bool TryReadLine(ref ReadOnlySequence<byte> buffer, out
ReadOnlySequence<byte> line)
{
    // Look for a EOL in the buffer.
    SequencePosition? position = buffer.PositionOf((byte)'\n');

    if (position == null)
    {
        line = default;
        return false;
    }

    // Skip the line + the \n.
    line = buffer.Slice(0, position.Value);
    buffer = buffer.Slice(buffer.GetPosition(1, position.Value));
    return true;
}

```

There are two loops:

- `FillPipeAsync` reads from the `Socket` and writes to the `PipeWriter`.
- `ReadPipeAsync` reads from the `PipeReader` and parses incoming lines.

There are no explicit buffers allocated. All buffer management is delegated to the `PipeReader` and `PipeWriter` implementations. Delegating buffer management makes it easier for consuming code to focus solely on the business logic.

In the first loop:

- `PipeWriter.GetMemory(Int32)` is called to get memory from the underlying writer.
- `PipeWriter.Advance(Int32)` is called to tell the `PipeWriter` how much data was written to the buffer.

- `PipeWriter.FlushAsync` is called to make the data available to the `PipeReader`.

In the second loop, the `PipeReader` consumes the buffers written by `PipeWriter`. The buffers come from the socket. The call to `PipeReader.ReadAsync`:

- Returns a `ReadResult` that contains two important pieces of information:
 - The data that was read in the form of `ReadOnlySequence<byte>`.
 - A boolean `IsCompleted` that indicates if the end of data (EOF) has been reached.

After finding the end of line (EOL) delimiter and parsing the line:

- The logic processes the buffer to skip what's already processed.
- `PipeReader.AdvanceTo` is called to tell the `PipeReader` how much data has been consumed and examined.

The reader and writer loops end by calling `Complete`. `Complete` lets the underlying Pipe release the memory it allocated.

Backpressure and flow control

Ideally, reading and parsing work together:

- The reading thread consumes data from the network and puts it in buffers.
- The parsing thread is responsible for constructing the appropriate data structures.

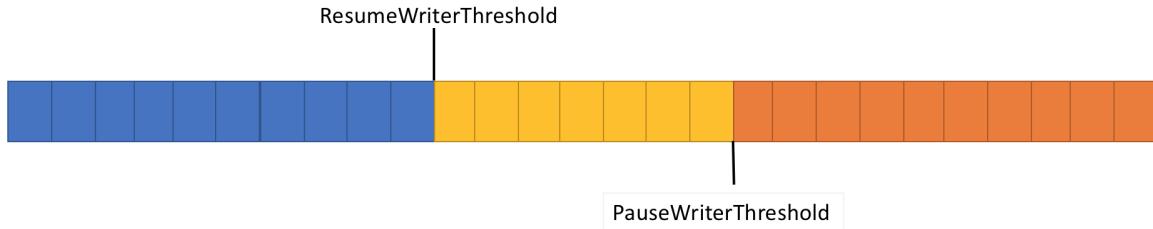
Typically, parsing takes more time than just copying blocks of data from the network:

- The reading thread gets ahead of the parsing thread.
- The reading thread has to either slow down or allocate more memory to store the data for the parsing thread.

For optimal performance, there's a balance between frequent pauses and allocating more memory.

To solve the preceding problem, the `Pipe` has two settings to control the flow of data:

- `PauseWriterThreshold`: Determines how much data should be buffered before calls to `FlushAsync` pause.
- `ResumeWriterThreshold`: Determines how much data the reader has to observe before calls to `PipeWriter.FlushAsync` resume.



PipeWriter.FlushAsync:

- Returns an incomplete `ValueTask<FlushResult>` when the amount of data in the `Pipe` crosses `PauseWriterThreshold`.
- Completes `ValueTask<FlushResult>` when it becomes lower than `ResumeWriterThreshold`.

Two values are used to prevent rapid cycling, which can occur if one value is used.

Examples

C#

```
// The Pipe will start returning incomplete tasks from FlushAsync until
// the reader examines at least 5 bytes.
var options = new PipeOptions(pauseWriterThreshold: 10,
resumeWriterThreshold: 5);
var pipe = new Pipe(options);
```

PipeScheduler

Typically when using `async` and `await`, asynchronous code resumes on either a `TaskScheduler` or the current `SynchronizationContext`.

When doing I/O, it's important to have fine-grained control over where the I/O is performed. This control allows taking advantage of CPU caches effectively. Efficient caching is critical for high-performance apps like web servers. `PipeScheduler` provides control over where asynchronous callbacks run. By default:

- The current `SynchronizationContext` is used.
- If there's no `SynchronizationContext`, it uses the thread pool to run callbacks.

C#

```
public static void Main(string[] args)
{
```

```

var writeScheduler = new SingleThreadPipeScheduler();
var readScheduler = new SingleThreadPipeScheduler();

// Tell the Pipe what schedulers to use and disable the
SynchronizationContext.
var options = new PipeOptions(readerScheduler: readScheduler,
                               writerScheduler: writeScheduler,
                               useSynchronizationContext: false);
var pipe = new Pipe(options);
}

// This is a sample scheduler that async callbacks on a single dedicated
thread.
public class SingleThreadPipeScheduler : PipeScheduler
{
    private readonly BlockingCollection<(Action<object> Action, object
State)> _queue =
    new BlockingCollection<(Action<object> Action, object State)>();
    private readonly Thread _thread;

    public SingleThreadPipeScheduler()
    {
        _thread = new Thread(DoWork);
        _thread.Start();
    }

    private void DoWork()
    {
        foreach (var item in _queue.GetConsumingEnumerable())
        {
            item.Action(item.State);
        }
    }

    public override void Schedule(Action<object?> action, object? state)
    {
        if (state is not null)
        {
            _queue.Add((action, state));
        }
        // else log the fact that _queue.Add was not called.
    }
}

```

`PipeScheduler.ThreadPool` is the `PipeScheduler` implementation that queues callbacks to the thread pool. `PipeScheduler.ThreadPool` is the default and generally the best choice. `PipeScheduler.Inline` can cause unintended consequences such as deadlocks.

Pipe reset

It's frequently efficient to reuse the `Pipe` object. To reset the pipe, call `PipeReader.Reset` when both the `PipeReader` and `PipeWriter` are complete.

PipeReader

`PipeReader` manages memory on the caller's behalf. Always call `PipeReader.AdvanceTo` after calling `PipeReader.ReadAsync`. This lets the `PipeReader` know when the caller is done with the memory so that it can be tracked. The `ReadOnlySequence<byte>` returned from `PipeReader.ReadAsync` is only valid until the call the `PipeReader.AdvanceTo`. It's illegal to use `ReadOnlySequence<byte>` after calling `PipeReader.AdvanceTo`.

`PipeReader.AdvanceTo` takes two `SequencePosition` arguments:

- The first argument determines how much memory was consumed.
- The second argument determines how much of the buffer was observed.

Marking data as consumed means that the pipe can return the memory to the underlying buffer pool. Marking data as observed controls what the next call to `PipeReader.ReadAsync` does. Marking everything as observed means that the next call to `PipeReader.ReadAsync` won't return until there's more data written to the pipe. Any other value will make the next call to `PipeReader.ReadAsync` return immediately with the observed *and* unobserved data, but not data that has already been consumed.

Read streaming data scenarios

There are a couple of typical patterns that emerge when trying to read streaming data:

- Given a stream of data, parse a single message.
- Given a stream of data, parse all available messages.

The following examples use the `TryParseLines` method for parsing messages from a `ReadOnlySequence<byte>`. `TryParseLines` parses a single message and updates the input buffer to trim the parsed message from the buffer. `TryParseLines` isn't part of .NET, it's a user written method used in the following sections.

C#

```
bool TryParseLines(ref ReadOnlySequence<byte> buffer, out Message message);
```

Read a single message

The following code reads a single message from a `PipeReader` and returns it to the caller.

C#

```
async ValueTask<Message?> ReadSingleMessageAsync(PipeReader reader,
    CancellationToken cancellationToken = default)
{
    while (true)
    {
        ReadResult result = await reader.ReadAsync(cancellationToken);
        ReadOnlySequence<byte> buffer = result.Buffer;

        // In the event that no message is parsed successfully, mark
        consumed
        // as nothing and examined as the entire buffer.
        SequencePosition consumed = buffer.Start;
        SequencePosition examined = buffer.End;

        try
        {
            if (TryParseLines(ref buffer, out Message message))
            {
                // A single message was successfully parsed so mark the
                start of the
                // parsed buffer as consumed. TryParseLines trims the buffer
                to
                // point to the data after the message was parsed.
                consumed = buffer.Start;

                // Examined is marked the same as consumed here, so the next
                call
                // to ReadSingleMessageAsync will process the next message
                if there's
                // one.
                examined = consumed;

                return message;
            }

            // There's no more data to be processed.
            if (result.IsCompleted)
            {
                if (buffer.Length > 0)
                {
                    // The message is incomplete and there's no more data to
                    process.
                    throw new InvalidDataException("Incomplete message.");
                }

                break;
            }
        }
        finally
```

```
        {
            reader.AdvanceTo(consumed, examined);
        }
    }

    return null;
}
```

The preceding code:

- Parses a single message.
- Updates the consumed `SequencePosition` and examined `SequencePosition` to point to the start of the trimmed input buffer.

The two `SequencePosition` arguments are updated because `TryParseLines` removes the parsed message from the input buffer. Generally, when parsing a single message from the buffer, the examined position should be one of the following:

- The end of the message.
- The end of the received buffer if no message was found.

The single message case has the most potential for errors. Passing the wrong values to `examined` can result in an out of memory exception or an infinite loop. For more information, see the [PipeReader common problems](#) section in this article.

Reading multiple messages

The following code reads all messages from a `PipeReader` and calls `ProcessMessageAsync` on each.

C#

```
async Task ProcessMessagesAsync(PipeReader reader, CancellationToken cancellationToken = default)
{
    try
    {
        while (true)
        {
            ReadResult result = await reader.ReadAsync(cancellationToken);
            ReadOnlySequence<byte> buffer = result.Buffer;

            try
            {
                // Process all messages from the buffer, modifying the input
                // buffer on each
                // iteration.
                while (TryParseLines(ref buffer, out Message message))

```

```

        {
            await ProcessMessageAsync(message);
        }

        // There's no more data to be processed.
        if (result.IsCompleted)
        {
            if (buffer.Length > 0)
            {
                // The message is incomplete and there's no more
                data to process.
                throw new InvalidDataException("Incomplete
message.");
            }
            break;
        }
    }
    finally
    {
        // Since all messages in the buffer are being processed, you
        can use the
        // remaining buffer's Start and End position to determine
        consumed and examined.
        reader.AdvanceTo(buffer.Start, buffer.End);
    }
}
finally
{
    await reader.CompleteAsync();
}
}

```

Cancellation

PipeReader.ReadAsync:

- Supports passing a `CancellationToken`.
- Throws an `OperationCanceledException` if the `CancellationToken` is canceled while there's a read pending.
- Supports a way to cancel the current read operation via `PipeReader.CancelPendingRead`, which avoids raising an exception. Calling `PipeReader.CancelPendingRead` causes the current or next call to `PipeReader.ReadAsync` to return a `ReadResult` with `IsCanceled` set to `true`. This can be useful for halting the existing read loop in a non-destructive and non-exceptional way.

```
private PipeReader reader;

public MyConnection(PipeReader reader)
{
    this.reader = reader;
}

public void Abort()
{
    // Cancel the pending read so the process loop ends without an
    // exception.
    reader.CancelPendingRead();
}

public async Task ProcessMessagesAsync()
{
    try
    {
        while (true)
        {
            ReadResult result = await reader.ReadAsync();
            ReadOnlySequence<byte> buffer = result.Buffer;

            try
            {
                if (result.IsCanceled)
                {
                    // The read was canceled. You can quit without reading
                    // the existing data.
                    break;
                }

                // Process all messages from the buffer, modifying the input
                // buffer on each
                // iteration.
                while (TryParseLines(ref buffer, out Message message))
                {
                    await ProcessMessageAsync(message);
                }

                // There's no more data to be processed.
                if (result.IsCompleted)
                {
                    break;
                }
            }
            finally
            {
                // Since all messages in the buffer are being processed, you
                // can use the
                // remaining buffer's Start and End position to determine
                // consumed and examined.
                reader.AdvanceTo(buffer.Start, buffer.End);
            }
        }
    }
}
```

```
        }
    }
    finally
    {
        await reader.CompleteAsync();
    }
}
```

PipeReader common problems

- Passing the wrong values to `consumed` or `examined` may result in reading already read data.
- Passing `buffer.End` as examined may result in:
 - Stalled data
 - Possibly an eventual Out of Memory (OOM) exception if data isn't consumed.
For example, `PipeReader.AdvanceTo(position, buffer.End)` when processing a single message at a time from the buffer.
- Passing the wrong values to `consumed` or `examined` may result in an infinite loop.
For example, `PipeReader.AdvanceTo(buffer.Start)` if `buffer.Start` hasn't changed will cause the next call to `PipeReader.ReadAsync` to return immediately before new data arrives.
- Passing the wrong values to `consumed` or `examined` may result in infinite buffering (eventual OOM).
- Using the `ReadOnlySequence<byte>` after calling `PipeReader.AdvanceTo` may result in memory corruption (use after free).
- Failing to call `PipeReader.Complete/CompleteAsync` may result in a memory leak.
- Checking `ReadResult.IsCompleted` and exiting the reading logic before processing the buffer results in data loss. The loop exit condition should be based on `ReadResult.Buffer.IsEmpty` and `ReadResult.IsCompleted`. Doing this incorrectly could result in an infinite loop.

Problematic code

✖ Data loss

The `ReadResult` can return the final segment of data when `IsCompleted` is set to `true`. Not reading that data before exiting the read loop will result in data loss.

⚠ Warning

Do **NOT** use the following code. Using this sample will result in data loss, hangs, security issues and should **NOT** be copied. The following sample is provided to explain **PipeReader Common problems**.

C#

```
Environment.FailFast("This code is terrible, don't use it!");
while (true)
{
    ReadResult result = await reader.ReadAsync(cancellationToken);
    ReadOnlySequence<byte> dataLossBuffer = result.Buffer;

    if (result.IsCompleted)
        break;

    Process(ref dataLossBuffer, out Message message);

    reader.AdvanceTo(dataLossBuffer.Start, dataLossBuffer.End);
}
```

⚠ Warning

Do **NOT** use the preceding code. Using this sample will result in data loss, hangs, security issues and should **NOT** be copied. The preceding sample is provided to explain **PipeReader Common problems**.

✖ Infinite loop

The following logic may result in an infinite loop if the `Result.IsCompleted` is `true` but there's never a complete message in the buffer.

⚠ Warning

Do **NOT** use the following code. Using this sample will result in data loss, hangs, security issues and should **NOT** be copied. The following sample is provided to explain **PipeReader Common problems**.

C#

```
Environment.FailFast("This code is terrible, don't use it!");
while (true)
```

```
{  
    ReadResult result = await reader.ReadAsync(cancellationToken);  
    ReadOnlySequence<byte> infiniteLoopBuffer = result.Buffer;  
    if (result.IsCompleted && infiniteLoopBuffer.IsEmpty)  
        break;  
  
    Process(ref infiniteLoopBuffer, out Message message);  
  
    reader.AdvanceTo(infiniteLoopBuffer.Start, infiniteLoopBuffer.End);  
}
```

⚠ Warning

Do **NOT** use the preceding code. Using this sample will result in data loss, hangs, security issues and should **NOT** be copied. The preceding sample is provided to explain [PipeReader Common problems](#).

Here's another piece of code with the same problem. It's checking for a non-empty buffer before checking `ReadResult.IsCompleted`. Because it's in an `else if`, it will loop forever if there's never a complete message in the buffer.

⚠ Warning

Do **NOT** use the following code. Using this sample will result in data loss, hangs, security issues and should **NOT** be copied. The following sample is provided to explain [PipeReader Common problems](#).

C#

```
Environment.FailFast("This code is terrible, don't use it!");  
while (true)  
{  
    ReadResult result = await reader.ReadAsync(cancellationToken);  
    ReadOnlySequence<byte> infiniteLoopBuffer = result.Buffer;  
  
    if (!infiniteLoopBuffer.IsEmpty)  
        Process(ref infiniteLoopBuffer, out Message message);  
  
    else if (result.IsCompleted)  
        break;  
  
    reader.AdvanceTo(infiniteLoopBuffer.Start, infiniteLoopBuffer.End);  
}
```

⚠ Warning

Do NOT use the preceding code. Using this sample will result in data loss, hangs, security issues and should NOT be copied. The preceding sample is provided to explain **PipeReader Common problems**.

✖ Unresponsive application

Unconditionally calling `PipeReader.AdvanceTo` with `buffer.End` in the `examined` position may result in the application becoming unresponsive when parsing a single message. The next call to `PipeReader.AdvanceTo` won't return until:

- There's more data written to the pipe.
- And the new data wasn't previously examined.

⚠ Warning

Do NOT use the following code. Using this sample will result in data loss, hangs, security issues and should NOT be copied. The following sample is provided to explain **PipeReader Common problems**.

C#

```
Environment.FailFast("This code is terrible, don't use it!");
while (true)
{
    ReadResult result = await reader.ReadAsync(cancellationToken);
    ReadOnlySequence<byte> hangBuffer = result.Buffer;

    Process(ref hangBuffer, out Message message);

    if (result.IsCompleted)
        break;

    reader.AdvanceTo(hangBuffer.Start, hangBuffer.End);

    if (message != null)
        return message;
}
```

⚠ Warning

Do NOT use the preceding code. Using this sample will result in data loss, hangs, security issues and should NOT be copied. The preceding sample is provided to explain **PipeReader Common problems**.

✖ Out of Memory (OOM)

With the following conditions, the following code keeps buffering until an [OutOfMemoryException](#) occurs:

- There's no maximum message size.
- The data returned from the `PipeReader` doesn't make a complete message. For example, it doesn't make a complete message because the other side is writing a large message (For example, a 4-GB message).

⚠ Warning

Do **NOT** use the following code. Using this sample will result in data loss, hangs, security issues and should **NOT** be copied. The following sample is provided to explain [PipeReader Common problems](#).

C#

```
Environment.FailFast("This code is terrible, don't use it!");
while (true)
{
    ReadResult result = await reader.ReadAsync(cancellationToken);
    ReadOnlySequence<byte> thisCouldOutOfMemory = result.Buffer;

    Process(ref thisCouldOutOfMemory, out Message message);

    if (result.IsCompleted)
        break;

    reader.AdvanceTo(thisCouldOutOfMemory.Start, thisCouldOutOfMemory.End);

    if (message != null)
        return message;
}
```

⚠ Warning

Do **NOT** use the preceding code. Using this sample will result in data loss, hangs, security issues and should **NOT** be copied. The preceding sample is provided to explain [PipeReader Common problems](#).

✖ Memory Corruption

When writing helpers that read the buffer, any returned payload should be copied before calling `Advance`. The following example will return memory that the `Pipe` has

discarded and may reuse it for the next operation (read/write).

⚠ Warning

Do **NOT** use the following code. Using this sample will result in data loss, hangs, security issues and should **NOT** be copied. The following sample is provided to explain **PipeReader Common problems**.

C#

```
public class Message
{
    public ReadOnlySequence<byte> CorruptedPayload { get; set; }
}
```

C#

```
Environment.FailFast("This code is terrible, don't use it!");
Message message = null;

while (true)
{
    ReadResult result = await reader.ReadAsync(cancellationToken);
    ReadOnlySequence<byte> buffer = result.Buffer;

    ReadHeader(ref buffer, out int length);

    if (length <= buffer.Length)
    {
        message = new Message
        {
            // Slice the payload from the existing buffer
            CorruptedPayload = buffer.Slice(0, length)
        };

        buffer = buffer.Slice(length);
    }

    if (result.IsCompleted)
        break;

    reader.AdvanceTo(buffer.Start, buffer.End);

    if (message != null)
    {
        // This code is broken since reader.AdvanceTo() was called with
        // a position *after* the buffer
        // was captured.
        break;
    }
}
```

```
    }

    return message;
}
```

⚠ Warning

Do NOT use the preceding code. Using this sample will result in data loss, hangs, security issues and should NOT be copied. The preceding sample is provided to explain **PipeReader Common problems**.

PipeWriter

The [PipeWriter](#) manages buffers for writing on the caller's behalf. `PipeWriter` implements `IBufferWriter<byte>`. `IBufferWriter<byte>` makes it possible to get access to buffers to perform writes without extra buffer copies.

C#

```
async Task WriteHelloAsync(PipeWriter writer, CancellationToken cancellationToken = default)
{
    // Request at least 5 bytes from the PipeWriter.
    Memory<byte> memory = writer.GetMemory(5);

    // Write directly into the buffer.
    int written = Encoding.ASCII.GetBytes("Hello".AsSpan(), memory.Span);

    // Tell the writer how many bytes were written.
    writer.Advance(written);

    await writer.FlushAsync(cancellationToken);
}
```

The previous code:

- Requests a buffer of at least 5 bytes from the `PipeWriter` using [GetMemory](#).
- Writes bytes for the ASCII string `"Hello"` to the returned `Memory<byte>`.
- Calls [Advance](#) to indicate how many bytes were written to the buffer.
- Flushes the `PipeWriter`, which sends the bytes to the underlying device.

The previous method of writing uses the buffers provided by the `PipeWriter`. It could also have used [PipeWriter.WriteAsync](#), which:

- Copies the existing buffer to the `PipeWriter`.
- Calls `GetSpan`, `Advance` as appropriate and calls `FlushAsync`.

C#

```
async Task WriteHelloAsync(PipeWriter writer, CancellationToken cancellationToken = default)
{
    byte[] helloBytes = Encoding.ASCII.GetBytes("Hello");

    // Write helloBytes to the writer, there's no need to call Advance here
    // (Write does that).
    await writer.WriteAsync(helloBytes, cancellationToken);
}
```

Cancellation

`FlushAsync` supports passing a `CancellationToken`. Passing a `CancellationToken` results in an `OperationCanceledException` if the token is canceled while there's a flush pending. `PipeWriter.FlushAsync` supports a way to cancel the current flush operation via `PipeWriter.CancelPendingFlush` without raising an exception. Calling `PipeWriter.CancelPendingFlush` causes the current or next call to `PipeWriter.FlushAsync` or `PipeWriter.WriteAsync` to return a `FlushResult` with `IsCanceled` set to `true`. This can be useful for halting the yielding flush in a non-destructive and non-exceptional way.

PipeWriter common problems

- `GetSpan` and `GetMemory` return a buffer with at least the requested amount of memory. **Don't** assume exact buffer sizes.
- There's no guarantee that successive calls will return the same buffer or the same-sized buffer.
- A new buffer must be requested after calling `Advance` to continue writing more data. The previously acquired buffer can't be written to.
- Calling `GetMemory` or `GetSpan` while there's an incomplete call to `FlushAsync` isn't safe.
- Calling `Complete` or `CompleteAsync` while there's unflushed data can result in memory corruption.

Tips for using PipeReader and PipeWriter

The following tips will help you use the `System.IO.Pipelines` classes successfully:

- Always complete the `PipeReader` and `PipeWriter`, including an exception where applicable.
- Always call `PipeReader.AdvanceTo` after calling `PipeReader.ReadAsync`.
- Periodically `await PipeWriter.FlushAsync` while writing, and always check `FlushResult.IsCompleted`. Abort writing if `IsCompleted` is `true`, as that indicates the reader is completed and no longer cares about what is written.
- Do call `PipeWriter.FlushAsync` after writing something that you want the `PipeReader` to have access to.
- Do not call `FlushAsync` if the reader can't start until `FlushAsync` finishes, as that may cause a deadlock.
- Ensure that only one context "owns" a `PipeReader` or `PipeWriter` or accesses them. These types are not thread-safe.
- Never access a `ReadResult.Buffer` after calling `AdvanceTo` or completing the `PipeReader`.

IDuplexPipe

The `IDuplexPipe` is a contract for types that support both reading and writing. For example, a network connection would be represented by an `IDuplexPipe`.

Unlike `Pipe`, which contains a `PipeReader` and a `PipeWriter`, `IDuplexPipe` represents a single side of a full duplex connection. That means what is written to the `PipeWriter` will not be read from the `PipeReader`.

Streams

When reading or writing stream data, you typically read data using a de-serializer and write data using a serializer. Most of these read and write stream APIs have a `Stream` parameter. To make it easier to integrate with these existing APIs, `PipeReader` and `PipeWriter` expose an `AsStream` method. `AsStream` returns a `Stream` implementation around the `PipeReader` or `PipeWriter`.

Stream example

`PipeReader` and `PipeWriter` instances can be created using the static `Create` methods given a `Stream` object and optional corresponding creation options.

The `StreamPipeReaderOptions` allow for control over the creation of the `PipeReader` instance with the following parameters:

- `StreamPipeReaderOptions.BufferSize` is the minimum buffer size in bytes used when renting memory from the pool, and defaults to `4096`.
- `StreamPipeReaderOptions.LeaveOpen` flag determines whether or not the underlying stream is left open after the `PipeReader` completes, and defaults to `false`.
- `StreamPipeReaderOptions.MinimumReadSize` represents the threshold of remaining bytes in the buffer before a new buffer is allocated, and defaults to `1024`.
- `StreamPipeReaderOptions.Pool` is the `MemoryPool<byte>` used when allocating memory, and defaults to `null`.

The `StreamPipeWriterOptions` allow for control over the creation of the `PipeWriter` instance with the following parameters:

- `StreamPipeWriterOptions.LeaveOpen` flag determines whether or not the underlying stream is left open after the `PipeWriter` completes, and defaults to `false`.
- `StreamPipeWriterOptions.MinimumBufferSize` represents the minimum buffer size to use when renting memory from the `Pool`, and defaults to `4096`.
- `StreamPipeWriterOptions.Pool` is the `MemoryPool<byte>` used when allocating memory, and defaults to `null`.

Important

When creating `PipeReader` and `PipeWriter` instances using the `Create` methods, you need to consider the `Stream` object lifetime. If you need access to the stream after the reader or writer is done with it, you'll need to set the `LeaveOpen` flag to `true` on the creation options. Otherwise, the stream will be closed.

The following code demonstrates the creation of `PipeReader` and `PipeWriter` instances using the `Create` methods from a stream.

C#

```
using System.Buffers;
using System.IO.Pipelines;
using System.Text;

class Program
{
    static async Task Main()
    {
```

```
using var stream = File.OpenRead("lorem-ipsum.txt");

var reader = PipeReader.Create(stream);
var writer = PipeWriter.Create(
    Console.OpenStandardOutput(),
    new StreamPipeWriterOptions(leaveOpen: true));

WriteUserCancellationPrompt();

var processMessagesTask = ProcessMessagesAsync(reader, writer);
var userCanceled = false;
var cancelProcessingTask = Task.Run(() =>
{
    while (char.ToUpperInvariant(Console.ReadKey().KeyChar) != 'C')
    {
        WriteUserCancellationPrompt();
    }

    userCanceled = true;

    // No exceptions thrown
    reader.CancelPendingRead();
    writer.CancelPendingFlush();
});

await Task.WhenAny(cancelProcessingTask, processMessagesTask);

Console.WriteLine(
    $"\\n\\nProcessing {(userCanceled ? "cancelled" : "completed")}.\\n");
}

static void WriteUserCancellationPrompt() =>
Console.WriteLine("Press 'C' to cancel processing...\\n");

static async Task ProcessMessagesAsync(
    PipeReader reader,
    PipeWriter writer)
{
    try
    {
        while (true)
        {
            ReadResult readResult = await reader.ReadAsync();
            ReadOnlySequence<byte> buffer = readResult.Buffer;

            try
            {
                if (readResult.IsCanceled)
                {
                    break;
                }

                if (TryParseLines(ref buffer, out string message))
                {

```

```

        FlushResult flushResult =
            await WriteMessagesAsync(writer, message);

        if (flushResult.IsCanceled ||
flushResult.IsCompleted)
        {
            break;
        }

        if (readResult.IsCompleted)
        {
            if (!buffer.IsEmpty)
            {
                throw new InvalidDataException("Incomplete
message.");
            }
            break;
        }
    }
    finally
    {
        reader.AdvanceTo(buffer.Start, buffer.End);
    }
}
}

catch (Exception ex)
{
    Console.Error.WriteLine(ex);
}
finally
{
    await reader.CompleteAsync();
    await writer.CompleteAsync();
}
}

static bool TryParseLines(
    ref ReadOnlySequence<byte> buffer,
    out string message)
{
    SequencePosition? position;
    StringBuilder outputMessage = new();

    while(true)
    {
        position = buffer.PositionOf((byte)'\n');

        if (!position.HasValue)
            break;

        outputMessage.Append(Encoding.ASCII.GetString(buffer.Slice(buffer.Start,
position.Value)))
                    .AppendLine();
    }
}

```

```
        buffer = buffer.Slice(buffer.GetPosition(1, position.Value));
    };

    message = outputMessage.ToString();
    return message.Length != 0;
}

static ValueTask<FlushResult> WriteMessagesAsync(
    PipeWriter writer,
    string message) =>
    writer.WriteAsync(Encoding.ASCII.GetBytes(message));
}
```

The application uses a [StreamReader](#) to read the *lorem-ipsum.txt* file as a stream, and it must end with a blank line. The [FileStream](#) is passed to [PipeReader.Create](#), which instantiates a [PipeReader](#) object. The console application then passes its standard output stream to [PipeWriter.Create](#) using [Console.OpenStandardOutput\(\)](#). The example supports [cancellation](#).

 [Collaborate with us on GitHub](#)

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



[.NET feedback](#)

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

Work with Buffers in .NET

Article • 02/08/2023

This article provides an overview of types that help read data that runs across multiple buffers. They're primarily used to support [PipeReader](#) objects.

IBufferWriter<T>

[System.Buffers.IBufferWriter<T>](#) is a contract for synchronous buffered writing. At the lowest level, the interface:

- Is basic and not difficult to use.
- Allows access to a [Memory<T>](#) or [Span<T>](#). The [Memory<T>](#) or [Span<T>](#) can be written to and you can determine how many [T](#) items were written.

C#

```
void WriteHello(IBufferWriter<byte> writer)
{
    // Request at least 5 bytes.
    Span<byte> span = writer.GetSpan(5);
    ReadOnlySpan<char> helloSpan = "Hello".AsSpan();
    int written = Encoding.ASCII.GetBytes(helloSpan, span);

    // Tell the writer how many bytes were written.
    writer.Advance(written);
}
```

The preceding method:

- Requests a buffer of at least 5 bytes from the [IBufferWriter<byte>](#) using [GetSpan\(5\)](#).
- Writes bytes for the ASCII string "Hello" to the returned [Span<byte>](#).
- Calls [IBufferWriter<T>](#) to indicate how many bytes were written to the buffer.

This method of writing uses the [Memory<T>/Span<T>](#) buffer provided by the [IBufferWriter<T>](#). Alternatively, the [Write](#) extension method can be used to copy an existing buffer to the [IBufferWriter<T>](#). [Write](#) does the work of calling [GetSpan](#)/[Advance](#) as appropriate, so there's no need to call [Advance](#) after writing:

C#

```

void WriteHello(IBufferWriter<byte> writer)
{
    byte[] helloBytes = Encoding.ASCII.GetBytes("Hello");

    // Write helloBytes to the writer. There's no need to call Advance here
    // since Write calls Advance.
    writer.Write(helloBytes);
}

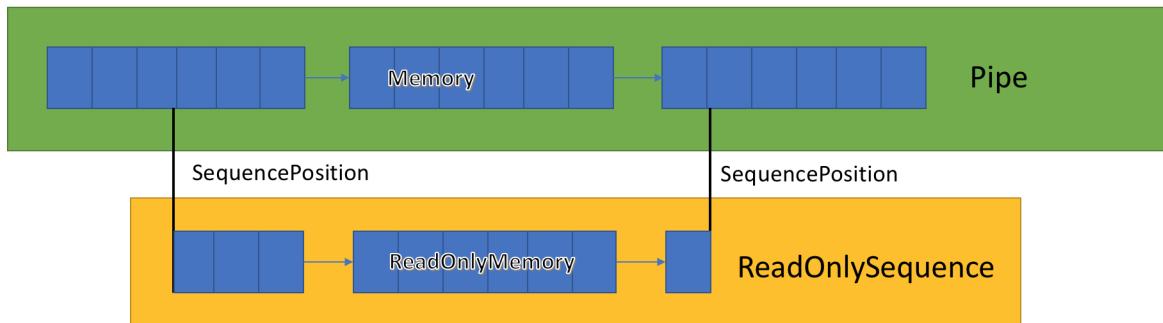
```

`ArrayBufferWriter<T>` is an implementation of `IBufferWriter<T>` whose backing store is a single contiguous array.

IBufferWriter common problems

- `GetSpan` and `GetMemory` return a buffer with at least the requested amount of memory. Don't assume exact buffer sizes.
- There's no guarantee that successive calls will return the same buffer or the same-sized buffer.
- A new buffer must be requested after calling `Advance` to continue writing more data. A previously acquired buffer cannot be written to after `Advance` has been called.

ReadOnlySequence<T>



`ReadOnlySequence<T>` is a struct that can represent a contiguous or noncontiguous sequence of `T`. It can be constructed from:

1. A `T[]`
2. A `ReadOnlyMemory<T>`
3. A pair of linked list node `ReadOnlySequenceSegment<T>` and index to represent the start and end position of the sequence.

The third representation is the most interesting one as it has performance implications on various operations on the `ReadOnlySequence<T>`:

| Representation | Operation | Complexity |
|---|--|--|
| <code>T[] / ReadOnlyMemory<T></code> | <code>Length</code> | <code>O(1)</code> |
| <code>T[] / ReadOnlyMemory<T></code> | <code>GetPosition(long)</code> | <code>O(1)</code> |
| <code>T[] / ReadOnlyMemory<T></code> | <code>Slice(int, int)</code> | <code>O(1)</code> |
| <code>T[] / ReadOnlyMemory<T></code> | <code>Slice(SequencePosition, SequencePosition)</code> | <code>O(1)</code> |
| <code>ReadOnlySequenceSegment<T></code> | <code>Length</code> | <code>O(1)</code> |
| <code>ReadOnlySequenceSegment<T></code> | <code>GetPosition(long)</code> | <code>O(number of segments)</code> |
| <code>ReadOnlySequenceSegment<T></code> | <code>Slice(int, int)</code> | <code>O(number of segments)</code> |
| <code>ReadOnlySequenceSegment<T></code> | <code>Slice(SequencePosition, SequencePosition)</code> | <code>O(1)</code> |

Because of this mixed representation, the `ReadOnlySequence<T>` exposes indexes as `SequencePosition` instead of an integer. A `SequencePosition`:

- Is an opaque value that represents an index into the `ReadOnlySequence<T>` where it originated.
- Consists of two parts, an integer and an object. What these two values represent are tied to the implementation of `ReadOnlySequence<T>`.

Access data

The `ReadOnlySequence<T>` exposes data as an enumerable of `ReadOnlyMemory<T>`. Enumerating each of the segments can be done using a basic foreach:

C#

```
long FindIndexOf(in ReadOnlySequence<byte> buffer, byte data)
{
    long position = 0;

    foreach (ReadOnlyMemory<byte> segment in buffer)
    {
        ReadOnlySpan<byte> span = segment.Span;
```

```

        var index = span.IndexOf(data);
        if (index != -1)
        {
            return position + index;
        }

        position += span.Length;
    }

    return -1;
}

```

The preceding method searches each segment for a specific byte. If you need to keep track of each segment's `SequencePosition`, `ReadOnlySequence<T>.TryGet` is more appropriate. The next sample changes the preceding code to return a `SequencePosition` instead of an integer. Returning a `SequencePosition` has the benefit of allowing the caller to avoid a second scan to get the data at a specific index.

C#

```

SequencePosition? FindIndexOf(in ReadOnlySequence<byte> buffer, byte data)
{
    SequencePosition position = buffer.Start;
    SequencePosition result = position;

    while (buffer.TryGet(ref position, out ReadOnlyMemory<byte> segment))
    {
        ReadOnlySpan<byte> span = segment.Span;
        var index = span.IndexOf(data);
        if (index != -1)
        {
            return buffer.GetPosition(index, result);
        }

        result = position;
    }
    return null;
}

```

The combination of `SequencePosition` and `TryGet` acts like an enumerator. The `position` field is modified at the start of each iteration to be start of each segment within the `ReadOnlySequence<T>`.

The preceding method exists as an extension method on `ReadOnlySequence<T>`. `PositionOf` can be used to simplify the preceding code:

C#

```
SequencePosition? FindIndex0f(in ReadOnlySequence<byte> buffer, byte data)
=> buffer.PositionOf(data);
```

Process a `ReadOnlySequence<T>`

Processing a `ReadOnlySequence<T>` can be challenging since data may be split across multiple segments within the sequence. For the best performance, split code into two paths:

- A fast path that deals with the single segment case.
- A slow path that deals with the data split across segments.

There are a few approaches that can be used to process data in multi-segmented sequences:

- Use the `SequenceReader<T>`.
- Parse data segment by segment, keeping track of the `SequencePosition` and index within the segment parsed. This avoids unnecessary allocations but may be inefficient, especially for small buffers.
- Copy the `ReadOnlySequence<T>` to a contiguous array and treat it like a single buffer:
 - If the size of the `ReadOnlySequence<T>` is small, it may be reasonable to copy the data into a stack-allocated buffer using the `stackalloc` operator.
 - Copy the `ReadOnlySequence<T>` into a pooled array using `ArrayPool<T>.Shared`.
 - Use `ReadOnlySequence<T>.ToArray()`. This isn't recommended in hot paths as it allocates a new `T[]` on the heap.

The following examples demonstrate some common cases for processing

`ReadOnlySequence<byte>`:

Process binary data

The following example parses a 4-byte big-endian integer length from the start of the `ReadOnlySequence<byte>`.

C#

```
bool TryParseHeaderLength(ref ReadOnlySequence<byte> buffer, out int length)
{
    // If there's not enough space, the length can't be obtained.
    if (buffer.Length < 4)
    {
```

```

        length = 0;
        return false;
    }

    // Grab the first 4 bytes of the buffer.
    var lengthSlice = buffer.Slice(buffer.Start, 4);
    if (lengthSlice.IsSingleSegment)
    {
        // Fast path since it's a single segment.
        length =
BinaryPrimitives.ReadInt32BigEndian(lengthSlice.First.Span);
    }
    else
    {
        // There are 4 bytes split across multiple segments. Since it's so
        small, it
            // can be copied to a stack allocated buffer. This avoids a heap
        allocation.
        Span<byte> stackBuffer = stackalloc byte[4];
        lengthSlice.CopyTo(stackBuffer);
        length = BinaryPrimitives.ReadInt32BigEndian(stackBuffer);
    }

    // Move the buffer 4 bytes ahead.
    buffer = buffer.Slice(lengthSlice.End);

    return true;
}

```

Process text data

The following example:

- Finds the first newline (\r\n) in the `ReadOnlySequence<byte>` and returns it via the `out 'line'` parameter.
- Trims that line, excluding the `\r\n` from the input buffer.

C#

```

static bool TryParseLine(ref ReadOnlySequence<byte> buffer, out
ReadOnlySequence<byte> line)
{
    SequencePosition position = buffer.Start;
    SequencePosition previous = position;
    var index = -1;
    line = default;

    while (buffer.TryGet(ref position, out ReadOnlyMemory<byte> segment))
    {
        ReadOnlySpan<byte> span = segment.Span;

```

```

    // Look for \r in the current segment.
    index = span.IndexOf((byte)'\r');

    if (index != -1)
    {
        // Check next segment for \n.
        if (index + 1 >= span.Length)
        {
            var next = position;
            if (!buffer.TryGet(ref next, out ReadOnlyMemory<byte>
nextSegment))
            {
                // You're at the end of the sequence.
                return false;
            }
            else if (nextSegment.Span[0] == (byte)'\n')
            {
                // A match was found.
                break;
            }
        }
        // Check the current segment of \n.
        else if (span[index + 1] == (byte)'\n')
        {
            // It was found.
            break;
        }
    }

    previous = position;
}

if (index != -1)
{
    // Get the position just before the \r\n.
    var delimiter = buffer.GetPosition(index, previous);

    // Slice the line (excluding \r\n).
    line = buffer.Slice(buffer.Start, delimiter);

    // Slice the buffer to get the remaining data after the line.
    buffer = buffer.Slice(buffer.GetPosition(2, delimiter));
    return true;
}

return false;
}

```

Empty segments

It's valid to store empty segments inside of a `ReadOnlySequence<T>`. Empty segments may occur while enumerating segments explicitly:

C#

```
static void EmptySegments()
{
    // This logic creates a ReadOnlySequence<byte> with 4 segments,
    // two of which are empty.
    var first = new BufferSegment(new byte[0]);
    var last = first.Append(new byte[] { 97 })
        .Append(new byte[0]).Append(new byte[] { 98 });

    // Construct the ReadOnlySequence<byte> from the linked list segments.
    var data = new ReadOnlySequence<byte>(first, 0, last, 1);

    // Slice using numbers.
    var sequence1 = data.Slice(0, 2);

    // Slice using SequencePosition pointing at the empty segment.
    var sequence2 = data.Slice(data.Start, 2);

    Console.WriteLine($"sequence1.Length={sequence1.Length}"); // sequence1.Length=2
    Console.WriteLine($"sequence2.Length={sequence2.Length}"); // sequence2.Length=2

    // sequence1.FirstSpan.Length=1
    Console.WriteLine($"sequence1.FirstSpan.Length={sequence1.FirstSpan.Length}");

    // Slicing using SequencePosition will Slice the ReadOnlySequence<byte>
    // directly
    // on the empty segment!
    // sequence2.FirstSpan.Length=0
    Console.WriteLine($"sequence2.FirstSpan.Length={sequence2.FirstSpan.Length}");

    // The following code prints 0, 1, 0, 1.
    SequencePosition position = data.Start;
    while (data.TryGet(ref position, out ReadOnlyMemory<byte> memory))
    {
        Console.WriteLine(memory.Length);
    }
}

class BufferSegment : ReadOnlySequenceSegment<byte>
{
    public BufferSegment(Memory<byte> memory)
    {
        Memory = memory;
    }

    public BufferSegment Append(Memory<byte> memory)
    {
        var segment = new BufferSegment(memory)
        {
```

```

        RunningIndex = RunningIndex + Memory.Length
    };
    Next = segment;
    return segment;
}
}

```

The preceding code creates a `ReadOnlySequence<byte>` with empty segments and shows how those empty segments affect the various APIs:

- `ReadOnlySequence<T>.Slice` with a `SequencePosition` pointing to an empty segment preserves that segment.
- `ReadOnlySequence<T>.Slice` with an int skips over the empty segments.
- Enumerating the `ReadOnlySequence<T>` enumerates the empty segments.

Potential problems with `ReadOnlySequence<T>` and `SequencePosition`

There are several unusual outcomes when dealing with a `ReadOnlySequence<T>/SequencePosition` vs. a normal `ReadOnlySpan<T>/ReadOnlyMemory<T>/T[]/int`:

- `SequencePosition` is a position marker for a specific `ReadOnlySequence<T>`, not an absolute position. Because it's relative to a specific `ReadOnlySequence<T>`, it doesn't have meaning if used outside of the `ReadOnlySequence<T>` where it originated.
- Arithmetic can't be performed on `SequencePosition` without the `ReadOnlySequence<T>`. That means doing basic things like `position++` is written `position = ReadOnlySequence<T>.GetPosition(1, position)`.
- `GetPosition(long)` does **not** support negative indexes. That means it's impossible to get the second to last character without walking all segments.
- Two `SequencePosition` can't be compared, making it difficult to:
 - Know if one position is greater than or less than another position.
 - Write some parsing algorithms.
- `ReadOnlySequence<T>` is bigger than an object reference and should be passed by `in` or `ref` where possible. Passing `ReadOnlySequence<T>` by `in` or `ref` reduces copies of the `struct`.
- Empty segments:
 - Are valid within a `ReadOnlySequence<T>`.
 - Can appear when iterating using the `ReadOnlySequence<T>.TryGet` method.
 - Can appear slicing the sequence using the `ReadOnlySequence<T>.Slice()` method with `SequencePosition` objects.

SequenceReader<T>

SequenceReader<T>:

- Is a new type that was introduced in .NET Core 3.0 to simplify the processing of a `ReadOnlySequence<T>`.
- Unifies the differences between a single segment `ReadOnlySequence<T>` and multi-segment `ReadOnlySequence<T>`.
- Provides helpers for reading binary and text data (`byte` and `char`) that may or may not be split across segments.

There are built-in methods for dealing with processing both binary and delimited data. The following section demonstrates what those same methods look like with the `SequenceReader<T>`:

Access data

`SequenceReader<T>` has methods for enumerating data inside of the `ReadOnlySequence<T>` directly. The following code is an example of processing a `ReadOnlySequence<byte>` a `byte` at a time:

C#

```
while (reader.TryRead(out byte b))
{
    Process(b);
}
```

The `CurrentSpan` exposes the current segment's `Span`, which is similar to what was done in the method manually.

Use position

The following code is an example implementation of `FindIndexOf` using the `SequenceReader<T>`:

C#

```
SequencePosition? FindIndexOf(in ReadOnlySequence<byte> buffer, byte data)
{
    var reader = new SequenceReader<byte>(buffer);

    while (!reader.End)
```

```
{  
    // Search for the byte in the current span.  
    var index = reader.CurrentSpan.IndexOf(data);  
    if (index != -1)  
    {  
        // It was found, so advance to the position.  
        reader.Advance(index);  
  
        return reader.Position;  
    }  
    // Skip the current segment since there's nothing in it.  
    reader.Advance(reader.CurrentSpan.Length);  
}  
  
return null;  
}
```

Process binary data

The following example parses a 4-byte big-endian integer length from the start of the `ReadOnlySequence<byte>`.

C#

```
bool TryParseHeaderLength(ref ReadOnlySequence<byte> buffer, out int length)  
{  
    var reader = new SequenceReader<byte>(buffer);  
    return reader.TryReadBigEndian(out length);  
}
```

Process text data

C#

```
static ReadOnlySpan<byte> NewLine => new byte[] { (byte)'\\r', (byte)'\\n' };  
  
static bool TryParseLine(ref ReadOnlySequence<byte> buffer,  
                        out ReadOnlySequence<byte> line)  
{  
    var reader = new SequenceReader<byte>(buffer);  
  
    if (reader.TryReadTo(out line, NewLine))  
    {  
        buffer = buffer.Slice(reader.Position);  
  
        return true;  
    }  
  
    line = default;
```

```
    return false;  
}
```

SequenceReader<T> common problems

- Because `SequenceReader<T>` is a mutable struct, it should always be passed by [reference](#).
- `SequenceReader<T>` is a [ref struct](#) so it can only be used in synchronous methods and can't be stored in fields. For more information, see [Avoid allocations](#).
- `SequenceReader<T>` is optimized for use as a forward-only reader. `Rewind` is intended for small backups that can't be addressed utilizing other `Read`, `Peek`, and `IsNext` APIs.

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

Memory-mapped files

Article • 12/14/2022

A memory-mapped file contains the contents of a file in virtual memory. This mapping between a file and memory space enables an application, including multiple processes, to modify the file by reading and writing directly to the memory. You can use managed code to access memory-mapped files in the same way that native Windows functions access memory-mapped files, as described in [Managing Memory-Mapped Files](#).

There are two types of memory-mapped files:

- Persisted memory-mapped files

Persisted files are memory-mapped files that are associated with a source file on a disk. When the last process has finished working with the file, the data is saved to the source file on the disk. These memory-mapped files are suitable for working with extremely large source files.

- Non-persisted memory-mapped files

Non-persisted files are memory-mapped files that are not associated with a file on a disk. When the last process has finished working with the file, the data is lost and the file is reclaimed by garbage collection. These files are suitable for creating shared memory for inter-process communications (IPC).

Processes, Views, and Managing Memory

Memory-mapped files can be shared across multiple processes. Processes can map to the same memory-mapped file by using a common name that is assigned by the process that created the file.

To work with a memory-mapped file, you must create a view of the entire memory-mapped file or a part of it. You can also create multiple views to the same part of the memory-mapped file, thereby creating concurrent memory. For two views to remain concurrent, they have to be created from the same memory-mapped file.

Multiple views may also be necessary if the file is greater than the size of the application's logical memory space available for memory mapping (2 GB on a 32-bit computer).

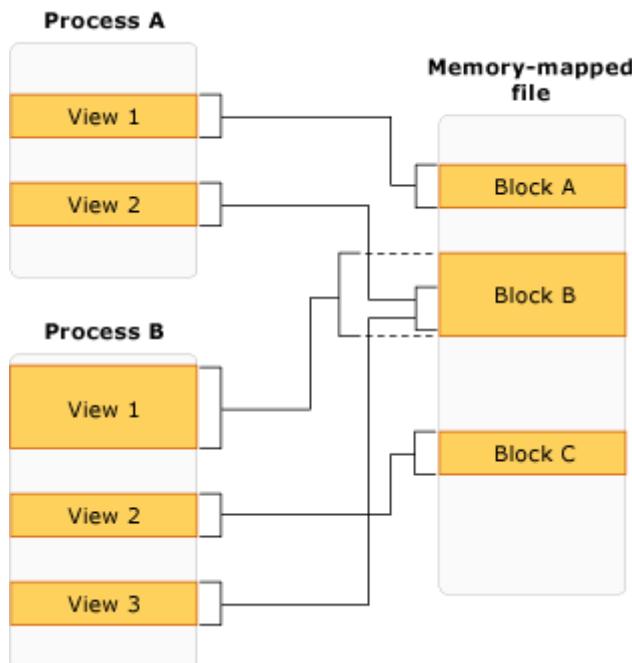
There are two types of views: stream access view and random access view. Use stream access views for sequential access to a file; this is recommended for non-persisted files

and IPC. Random access views are preferred for working with persisted files.

Memory-mapped files are accessed through the operating system's memory manager, so the file is automatically partitioned into a number of pages and accessed as needed. You do not have to handle the memory management yourself.

The following illustration shows how multiple processes can have multiple and overlapping views to the same memory-mapped file at the same time.

The following image shows multiple and overlapped views to a memory-mapped file:



Programming with Memory-Mapped Files

The following table provides a guide for using memory-mapped file objects and their members.

| Task | Methods or properties to use |
|--|---|
| To obtain a MemoryMappedFile object that represents a persisted memory-mapped file from a file on disk. | MemoryMappedFile.CreateFromFile method. |
| To obtain a MemoryMappedFile object that represents a non-persisted memory-mapped file (not associated with a file on disk). | MemoryMappedFile.CreateNew method. - or - MemoryMappedFile.CreateOrOpen method. |
| To obtain a MemoryMappedFile object of an existing memory- | MemoryMappedFile.OpenExisting method. |

| Task | Methods or properties to use |
|--|---|
| mapped file (either persisted or non-persisted). | |
| To obtain a UnmanagedMemoryStream object for a sequentially accessed view to the memory-mapped file. | MemoryMappedFile.CreateViewStream method. |
| To obtain a UnmanagedMemoryAccessor object for a random access view to a memory-mapped file. | MemoryMappedFile.CreateViewAccessor method. |
| To obtain a SafeMemoryMappedViewHandle object to use with unmanaged code. | MemoryMappedFile.SafeMemoryMappedFileHandle property. - or - |
| | MemoryMappedViewAccessor.SafeMemoryMappedViewHandle property. - or - |
| (To determine the current system page size, use the Environment.SystemPageSize property.) | CreateNew method with the MemoryMappedFileOptions.DelayAllocatePages value. - or - CreateOrOpen methods that have a MemoryMappedFileOptions enumeration as a parameter. |

Security

You can apply access rights when you create a memory-mapped file, by using the following methods that take a [MemoryMappedFileAccess](#) enumeration as a parameter:

- [MemoryMappedFile.CreateFromFile](#)
- [MemoryMappedFile.CreateNew](#)
- [MemoryMappedFile.CreateOrOpen](#)

You can specify access rights for opening an existing memory-mapped file by using the [OpenExisting](#) methods that take an [MemoryMappedFileRights](#) as a parameter.

In addition, you can include a [MemoryMappedFileSecurity](#) object that contains predefined access rules.

To apply new or changed access rules to a memory-mapped file, use the [SetAccessControl](#) method. To retrieve access or audit rules from an existing file, use the [GetAccessControl](#) method.

Examples

Persisted Memory-Mapped Files

The [CreateFromFile](#) methods create a memory-mapped file from an existing file on disk.

The following example creates a memory-mapped view of a part of an extremely large file and manipulates a portion of it.

C#

```
using System;
using System.IO;
using System.IO.MemoryMappedFiles;
using System.Runtime.InteropServices;

class Program
{
    static void Main(string[] args)
    {
        long offset = 0x10000000; // 256 megabytes
        long length = 0x20000000; // 512 megabytes

        // Create the memory-mapped file.
        using (var mmf =
MemoryMappedFile.CreateFromFile(@"c:\ExtremelyLargeImage.data",
 FileMode.Open, "ImgA"))
        {
            // Create a random access view, from the 256th megabyte (the
            offset)
            // to the 768th megabyte (the offset plus length).
            using (var accessor = mmf.CreateViewAccessor(offset, length))
            {
                int colorSize = Marshal.SizeOf(typeof(MyColor));
                MyColor color;

                // Make changes to the view.
                for (long i = 0; i < length; i += colorSize)
                {

```

```

                accessor.Read(i, out color);
                color.Brighten(10);
                accessor.Write(i, ref color);
            }
        }
    }
}

public struct MyColor
{
    public short Red;
    public short Green;
    public short Blue;
    public short Alpha;

    // Make the view brighter.
    public void Brighten(short value)
    {
        Red = (short)Math.Min(short.MaxValue, (int)Red + value);
        Green = (short)Math.Min(short.MaxValue, (int)Green + value);
        Blue = (short)Math.Min(short.MaxValue, (int)Blue + value);
        Alpha = (short)Math.Min(short.MaxValue, (int)Alpha + value);
    }
}

```

The following example opens the same memory-mapped file for another process.

C#

```

using System;
using System.IO.MemoryMappedFiles;
using System.Runtime.InteropServices;

class Program
{
    static void Main(string[] args)
    {
        // Assumes another process has created the memory-mapped file.
        using (var mmf = MemoryMappedFile.OpenExisting("ImgA"))
        {
            using (var accessor = mmf.CreateViewAccessor(4000000, 2000000))
            {
                int colorSize = Marshal.SizeOf(typeof(MyColor));
                MyColor color;

                // Make changes to the view.
                for (long i = 0; i < 1500000; i += colorSize)
                {
                    accessor.Read(i, out color);
                    color.Brighten(20);
                    accessor.Write(i, ref color);
                }
            }
        }
    }
}

```

```

        }
    }
}

public struct MyColor
{
    public short Red;
    public short Green;
    public short Blue;
    public short Alpha;

    // Make the view brighter.
    public void Brighten(short value)
    {
        Red = (short)Math.Min(short.MaxValue, (int)Red + value);
        Green = (short)Math.Min(short.MaxValue, (int)Green + value);
        Blue = (short)Math.Min(short.MaxValue, (int)Blue + value);
        Alpha = (short)Math.Min(short.MaxValue, (int)Alpha + value);
    }
}

```

Non-Persisted Memory-Mapped Files

The [CreateNew](#) and [CreateOrOpen](#) methods create a memory-mapped file that is not mapped to an existing file on disk.

The following example consists of three separate processes (console applications) that write Boolean values to a memory-mapped file. The following sequence of actions occur:

1. `Process A` creates the memory-mapped file and writes a value to it.
2. `Process B` opens the memory-mapped file and writes a value to it.
3. `Process C` opens the memory-mapped file and writes a value to it.
4. `Process A` reads and displays the values from the memory-mapped file.
5. After `Process A` is finished with the memory-mapped file, the file is immediately reclaimed by garbage collection.

To run this example, do the following:

1. Compile the applications and open three Command Prompt windows.
2. In the first Command Prompt window, run `Process A`.

3. In the second Command Prompt window, run **Process B**.

4. Return to **Process A** and press ENTER.

5. In the third Command Prompt window, run **Process C**.

6. Return to **Process A** and press ENTER.

The output of **Process A** is as follows:

Console

```
Start Process B and press ENTER to continue.  
Start Process C and press ENTER to continue.  
Process A says: True  
Process B says: False  
Process C says: True
```

Process A

C#

```
using System;  
using System.IO;  
using System.IO.MemoryMappedFiles;  
using System.Threading;  
  
class Program  
{  
    // Process A:  
    static void Main(string[] args)  
    {  
        using (MemoryMappedFile mmf = MemoryMappedFile.CreateNew("testmap",  
10000))  
        {  
            bool mutexCreated;  
            Mutex mutex = new Mutex(true, "testmapmutex", out mutexCreated);  
            using (MemoryMappedViewStream stream = mmf.CreateViewStream())  
            {  
                BinaryWriter writer = new BinaryWriter(stream);  
                writer.Write(1);  
            }  
            mutex.ReleaseMutex();  
  
            Console.WriteLine("Start Process B and press ENTER to  
continue.");  
            Console.ReadLine();  
  
            Console.WriteLine("Start Process C and press ENTER to  
continue.");  
            Console.ReadLine();
```

```

        mutex.WaitOne();
        using (MemoryMappedViewStream stream = mmf.CreateViewStream())
        {
            BinaryReader reader = new BinaryReader(stream);
            Console.WriteLine("Process A says: {0}",
reader.ReadBoolean());
            Console.WriteLine("Process B says: {0}",
reader.ReadBoolean());
            Console.WriteLine("Process C says: {0}",
reader.ReadBoolean());
        }
        mutex.ReleaseMutex();
    }
}

```

Process B

C#

```

using System;
using System.IO;
using System.IO.MemoryMappedFiles;
using System.Threading;

class Program
{
    // Process B:
    static void Main(string[] args)
    {
        try
        {
            using (MemoryMappedFile mmf =
MemoryMappedFile.OpenExisting("testmap"))
            {

                Mutex mutex = Mutex.OpenExisting("testmapmutex");
                mutex.WaitOne();

                using (MemoryMappedViewStream stream =
mmf.CreateViewStream(1, 0))
                {
                    BinaryWriter writer = new BinaryWriter(stream);
                    writer.Write(0);
                }
                mutex.ReleaseMutex();
            }
        }
        catch (FileNotFoundException)
        {
            Console.WriteLine("Memory-mapped file does not exist. Run
Process A first.");
        }
    }
}

```

```
        }
    }
}
```

Process C

C#

```
using System;
using System.IO;
using System.IO.MemoryMappedFiles;
using System.Threading;

class Program
{
    // Process C:
    static void Main(string[] args)
    {
        try
        {
            using (MemoryMappedFile mmf =
MemoryMappedFile.OpenExisting("testmap"))
            {

                Mutex mutex = Mutex.OpenExisting("testmapmutex");
                mutex.WaitOne();

                using (MemoryMappedViewStream stream =
mmf.CreateViewStream(2, 0))
                {
                    BinaryWriter writer = new BinaryWriter(stream);
                    writer.Write(1);
                }
                mutex.ReleaseMutex();
            }
        }
        catch (FileNotFoundException)
        {
            Console.WriteLine("Memory-mapped file does not exist. Run
Process A first, then B.");
        }
    }
}
```

See also

- [File and Stream I/O](#)

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

System.IO.FileStream class

Article • 01/08/2024

This article provides supplementary remarks to the reference documentation for this API.

Use the [FileStream](#) class to read from, write to, open, and close files on a file system, and to manipulate other file-related operating system handles, including pipes, standard input, and standard output. You can use the [Read](#), [Write](#), [CopyTo](#), and [Flush](#) methods to perform synchronous operations, or the [ReadAsync](#), [WriteAsync](#), [CopyToAsync](#), and [FlushAsync](#) methods to perform asynchronous operations. Use the asynchronous methods to perform resource-intensive file operations without blocking the main thread. This performance consideration is particularly important in a Windows 8.x Store app or desktop app where a time-consuming stream operation can block the UI thread and make your app appear as if it is not working. [FileStream](#) buffers input and output for better performance.

ⓘ Important

This type implements the [IDisposable](#) interface. When you have finished using the type, you should dispose of it either directly or indirectly. To dispose of the type directly, call its [Dispose](#) method in a `try/catch` block. To dispose of it indirectly, use a language construct such as `using` (in C#) or `Using` (in Visual Basic). For more information, see the "Using an Object that Implements [IDisposable](#)" section in the [IDisposable](#) interface topic.

The [IsAsync](#) property detects whether the file handle was opened asynchronously. You specify this value when you create an instance of the [FileStream](#) class using a constructor that has an `isAsync`, `useAsync`, or `options` parameter. When the property is `true`, the stream utilizes overlapped I/O to perform file operations asynchronously. However, the [IsAsync](#) property does not have to be `true` to call the [ReadAsync](#), [WriteAsync](#), or [CopyToAsync](#) method. When the [IsAsync](#) property is `false` and you call the asynchronous read and write operations, the UI thread is still not blocked, but the actual I/O operation is performed synchronously.

The [Seek](#) method supports random access to files. [Seek](#) allows the read/write position to be moved to any position within the file. This is done with byte offset reference point parameters. The byte offset is relative to the seek reference point, which can be the beginning, the current position, or the end of the underlying file, as represented by the three members of the [SeekOrigin](#) enumeration.

Note

Disk files always support random access. At the time of construction, the `CanSeek` property value is set to `true` or `false` depending on the underlying file type. If the underlying file type is `FILE_TYPE_DISK`, as defined in `winbase.h`, the `CanSeek` property value is `true`. Otherwise, the `CanSeek` property value is `false`.

If a process terminates with part of a file locked or closes a file that has outstanding locks, the behavior is undefined.

For directory operations and other file operations, see the [File](#), [Directory](#), and [Path](#) classes. The [File](#) class is a utility class that has static methods primarily for the creation of [FileStream](#) objects based on file paths. The [MemoryStream](#) class creates a stream from a byte array and is similar to the [FileStream](#) class.

For a list of common file and directory operations, see [Common I/O Tasks](#).

Detection of stream position changes

When a [FileStream](#) object does not have an exclusive hold on its handle, another thread could access the file handle concurrently and change the position of the operating system's file pointer that is associated with the file handle. In this case, the cached position in the [FileStream](#) object and the cached data in the buffer could be compromised. The [FileStream](#) object routinely performs checks on methods that access the cached buffer to ensure that the operating system's handle position is the same as the cached position used by the [FileStream](#) object.

If an unexpected change in the handle position is detected in a call to the [Read](#) method, .NET discards the contents of the buffer and reads the stream from the file again. This can affect performance, depending on the size of the file and any other processes that could affect the position of the file stream.

If an unexpected change in the handle position is detected in a call to the [Write](#) method, the contents of the buffer are discarded and an [IOException](#) exception is thrown.

A [FileStream](#) object will not have an exclusive hold on its handle when either the [SafeFileHandle](#) property is accessed to expose the handle or the [FileStream](#) object is given the [SafeFileHandle](#) property in its constructor.

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

.NET is an open source project.
Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

System.IO.FileSystemWatcher class

Article • 01/08/2024

This article provides supplementary remarks to the reference documentation for this API.

Use [FileSystemWatcher](#) to watch for changes in a specified directory. You can watch for changes in files and subdirectories of the specified directory. You can create a component to watch files on a local computer, a network drive, or a remote computer.

To watch for changes in all files, set the [Filter](#) property to an empty string ("") or use wildcards ("*.*"). To watch a specific file, set the [Filter](#) property to the file name. For example, to watch for changes in the file MyDoc.txt, set the [Filter](#) property to "MyDoc.txt". You can also watch for changes in a certain type of file. For example, to watch for changes in text files, set the [Filter](#) property to "*.txt".

There are several types of changes you can watch for in a directory or file. For example, you can watch for changes in [Attributes](#), the [LastWrite](#) date and time, or the [Size](#) of files or directories. This is done by setting the [NotifyFilter](#) property to one of the [NotifyFilters](#) values. For more information on the type of changes you can watch, see [NotifyFilters](#).

You can watch for renaming, deletion, or creation of files or directories. For example, to watch for renaming of text files, set the [Filter](#) property to "*.txt" and call the [WaitForChanged](#) method with a [Renamed](#) specified for its parameter.

The Windows operating system notifies your component of file changes in a buffer created by the [FileSystemWatcher](#). If there are many changes in a short time, the buffer can overflow. This causes the component to lose track of changes in the directory, and it will only provide blanket notification. Increasing the size of the buffer with the [InternalBufferSize](#) property is expensive, as it comes from non-paged memory that cannot be swapped out to disk, so keep the buffer as small yet large enough to not miss any file change events. To avoid a buffer overflow, use the [NotifyFilter](#) and [IncludeSubdirectories](#) properties so you can filter out unwanted change notifications.

For a list of initial property values for an instance of [FileSystemWatcher](#), see the [FileSystemWatcher](#) constructor.

Considerations when using the [FileSystemWatcher](#) class:

- Hidden files are not ignored.

- In some systems, [FileSystemWatcher](#) reports changes to files using the short 8.3 file name format. For example, a change to "LongFileName.LongExtension" could be reported as "LongFil~.Lon".
- This class contains a link demand and an inheritance demand at the class level that applies to all members. A [SecurityException](#) is thrown when either the immediate caller or the derived class does not have full-trust permission. For details about security demands, see [Link Demands](#).
- The maximum size you can set for the [InternalBufferSize](#) property for monitoring a directory over the network is 64 KB.

Copy and move folders

The operating system and [FileSystemWatcher](#) object interpret a cut-and-paste action or a move action as a rename action for a folder and its contents. If you cut and paste a folder with files into a folder being watched, the [FileSystemWatcher](#) object reports only the folder as new, but not its contents because they are essentially only renamed.

To be notified that the contents of folders have been moved or copied into a watched folder, provide [OnChanged](#) and [OnRenamed](#) event handler methods as suggested in the following table.

[+] [Expand table](#)

| Event Handler | Events Handled | Performs |
|---------------------------|---|---|
| OnChanged | Changed , Created , Deleted | Report changes in file attributes, created files, and deleted files. |
| OnRenamed | Renamed | List the old and new paths of renamed files and folders, expanding recursively if needed. |

Events and buffer sizes

Note that several factors can affect which file system change events are raised, as described by the following:

- Common file system operations might raise more than one event. For example, when a file is moved from one directory to another, several [OnChanged](#) and some [OnCreated](#) and [OnDeleted](#) events might be raised. Moving a file is a complex operation that consists of multiple simple operations, therefore raising multiple

events. Likewise, some applications (for example, antivirus software) might cause additional file system events that are detected by [FileSystemWatcher](#).

- The [FileSystemWatcher](#) can watch disks as long as they are not switched or removed. The [FileSystemWatcher](#) does not raise events for CDs and DVDs, because time stamps and properties cannot change. Remote computers must have one of the required platforms installed for the component to function properly.

Note that a [FileSystemWatcher](#) may miss an event when the buffer size is exceeded. To avoid missing events, follow these guidelines:

- Increase the buffer size by setting the [InternalBufferSize](#) property.
- Avoid watching files with long file names, because a long file name contributes to filling up the buffer. Consider renaming these files using shorter names.
- Keep your event handling code as short as possible.

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

.NET is an open source project.
Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

System.AppContext class

Article • 01/08/2024

This article provides supplementary remarks to the reference documentation for this API.

The [AppContext](#) class enables library writers to provide a uniform opt-out mechanism for new functionality for their users. It establishes a loosely coupled contract between components in order to communicate an opt-out request. This capability is typically important when a change is made to existing functionality. Conversely, there is already an implicit opt-in for new functionality.

AppContext for library developers

Libraries use the [AppContext](#) class to define and expose compatibility switches, while library users can set those switches to affect the library behavior. By default, libraries provide the new functionality, and they only alter it (that is, they provide the previous functionality) if the switch is set. This allows libraries to provide new behavior for an existing API while continuing to support callers who depend on the previous behavior.

Define the switch name

The most common way to allow consumers of your library to opt out of a change of behavior is to define a named switch. Its `value` element is a name/value pair that consists of the name of a switch and its [Boolean](#) value. By default, the switch is always implicitly `false`, which provides the new behavior (and makes the new behavior opt-in by default). Setting the switch to `true` enables it, which provides the legacy behavior. Explicitly setting the switch to `false` also provides the new behavior.

It's beneficial to use a consistent format for switch names, since they're a formal contract exposed by a library. The following are two obvious formats:

- *Switch.namespace.switchname*
- *Switch.library.switchname*

Once you define and document the switch, callers can use it by calling the [AppContext.SetSwitch\(String, Boolean\)](#) method programmatically. .NET Framework apps can also use the switch by adding an `<AppContextSwitchOverrides>` element to their application configuration file or by using the registry. For more information about how

callers use and set the value of [AppContext](#) configuration switches, see the [AppContext for library consumers](#) section.

In .NET Framework, when the common language runtime runs an application, it automatically reads the registry's compatibility settings and loads the application configuration file to populate the application's [AppContext](#) instance. Because the [AppContext](#) instance is populated either programmatically by the caller or by the runtime, .NET Framework apps don't have to take any action, such as calling the [SetSwitch](#) method, to configure the [AppContext](#) instance.

Check the setting

You can check if a consumer has declared the value of the switch and act appropriately by calling the [AppContext.TryGetSwitch](#) method. The method returns `true` if the `switchName` argument is found, and its `isEnabled` argument indicates the value of the switch. Otherwise, the method returns `false`.

Example

The following example illustrates the use of the [AppContext](#) class to allow the customer to choose the original behavior of a library method. The following is version 1.0 of a library named `StringLibrary`. It defines a `SubstringStartsAt` method that performs an ordinal comparison to determine the starting index of a substring within a larger string.

C#

```
using System;
using System.Reflection;

[assembly: AssemblyVersion("1.0.0.0")]

public static class StringLibrary1
{
    public static int SubstringStartsAt(string fullString, string substr)
    {
        return fullString.IndexOf(substr, StringComparison.Ordinal);
    }
}
```

The following example then uses the library to find the starting index of the substring "archæ" in "The archaeologist". Because the method performs an ordinal comparison, the substring cannot be found.

C#

```

using System;

public class Example1
{
    public static void Main()
    {
        string value = "The archaeologist";
        string substring = "archæ";
        int position = StringLibrary1.SubstringStartsAt(value, substring);
        if (position >= 0)
            Console.WriteLine("'{}' found in '{}' starting at position
{2}",
                            substring, value, position);
        else
            Console.WriteLine("'{}' not found in '{}'", substring, value);
    }
}

// The example displays the following output:
//      'archæ' not found in 'The archaeologist'

```

Version 2.0 of the library, however, changes the `SubstringStartsAt` method to use culture-sensitive comparison.

C#

```

using System;
using System.Reflection;

[assembly: AssemblyVersion("2.0.0.0")]

public static class StringLibrary2
{
    public static int SubstringStartsAt(string fullString, string substr)
    {
        return fullString.IndexOf(substr, StringComparison.CurrentCulture);
    }
}

```

When the app is recompiled to run against the new version of the library, it now reports that the substring "archæ" is found at index 4 in "The archaeologist".

C#

```

using System;

public class Example2
{
    public static void Main()
    {
        string value = "The archaeologist";

```

```

        string substring = "archæ";
        int position = StringLibrary2.SubstringStartsAt(value, substring);
        if (position >= 0)
            Console.WriteLine("'{0}' found in '{1}' starting at position
{2}",
                            substring, value, position);
        else
            Console.WriteLine("'{0}' not found in '{1}'", substring, value);
    }
}
// The example displays the following output:
//      'archæ' found in 'The archaeologist' starting at position 4

```

This change can be prevented from breaking the applications that depend on the original behavior by defining a switch. In this case, the switch is named

`StringLibrary.DoNotUseCultureSensitiveComparison`. Its default value, `false`, indicates that the library should perform its version 2.0 culture-sensitive comparison. `true` indicates that the library should perform its version 1.0 ordinal comparison. A slight modification of the previous code allows the library consumer to set the switch to determine the kind of comparison the method performs.

C#

```

using System;
using System.Reflection;

[assembly: AssemblyVersion("2.0.0.0")]

public static class StringLibrary
{
    public static int SubstringStartsAt(string fullString, string substr)
    {
        bool flag;
        if
(ApplicationContext.TryGetSwitch("StringLibrary.DoNotUseCultureSensitiveComparison",
out flag) && flag == true)
            return fullString.IndexOf(substr, StringComparison.Ordinal);
        else
            return fullString.IndexOf(substr, StringComparison.CurrentCulture);
    }
}

```

A .NET Framework application can then use the following configuration file to restore the version 1.0 behavior.

XML

```

<configuration>
  <runtime>

```

```
<AppContextSwitchOverrides  
value="StringLibrary.DoNotUseCultureSensitiveComparison=true" />  
</runtime>  
</configuration>
```

When the application is run with the configuration file present, it produces the following output:

Output

```
'archæ' not found in 'The archaeologist'
```

AppContext for library consumers

If you're the consumer of a library, the [AppContext](#) class allows you to take advantage of a library or library method's opt-out mechanism for new functionality. Individual methods of the class library that you are calling define particular switches that enable or disable a new behavior. The value of the switch is a Boolean. If it is `false`, which is typically the default value, the new behavior is enabled; if it is `true`, the new behavior is disabled, and the member behaves as it did previously.

You can set the value of a switch by calling the [AppContext.SetSwitch\(String, Boolean\)](#) method in your code. The `switchName` argument defines the switch name, and the `isEnabled` property defines the value of the switch. Because [AppContext](#) is a static class, it is available on a per-application domain basis. Calling the [AppContext.SetSwitch\(String, Boolean\)](#) has application scope; that is, it affects only the application.

.NET Framework apps have additional ways to set the value of a switch:

- By adding an `<AppContextSwitchOverrides>` element to the `<runtime>` section of the app.config file. The switch has a single attribute, `value`, whose value is a string that represents a key/value pair containing both the switch name and its value.

To define multiple switches, separate each switch's key/value pair in the `<AppContextSwitchOverrides>` element's `value` attribute with a semicolon. In that case, the `<AppContextSwitchOverrides>` element has the following format:

XML

```
<AppContextSwitchOverrides  
value="switchName1=value1;switchName2=value2" />
```

Using the `<AppContextSwitchOverrides>` element to define a configuration setting has application scope; that is, it affects only the application.

ⓘ Note

For information on the switches defined by .NET Framework, see `<AppContextSwitchOverrides>` element.

- By adding an entry to the registry. Add a new string value to the `HKLM\SOFTWARE\Microsoft\.NETFramework\AppBarContext` subkey. Set the name of the entry to the name of the switch. Set its value to one of the following options: `True`, `true`, `False`, or `false`. If the runtime encounters any other value, it ignores the switch.

On a 64-bit operating system, you must also add the same entry to the `HKLM\SOFTWARE\Wow6432Node\Microsoft\.NETFramework\AppBarContext` subkey.

Using the registry to define an `AppBarContext` switch has machine scope; that is, it affects every application running on the machine.

For ASP.NET and ASP.NET Core applications, you set a switch by adding an `<Add>` element to the `<appSettings>` section of the web.config file. For example:

XML

```
<appSettings>
  <add key="AppBarContext.SetSwitch:switchName1" value="switchValue1" />
  <add key="AppBarContext.SetSwitch:switchName2" value="switchValue2" />
</appSettings>
```

If you set the same switch in more than one way, the order of precedence for determining which setting overrides the others is:

1. The programmatic setting.
2. The setting in the app.config file (for .NET Framework apps) or the web.config file (for ASP.NET Core apps).
3. The registry setting (for .NET Framework apps only).

The following is a simple application that passes a file URI to the `Path.GetDirectoryName` method. It throws an `ArgumentException` because `file://` is no longer a valid part of a file path.

C#

```
using System;
using System.IO;
using System.Runtime.Versioning;

// [assembly: TargetFramework(".NETFramework, Version=v4.6.2")]

public class Example
{
    public static void Main()
    {

        Console.WriteLine(Path.GetDirectoryName("file://c/temp/dirlist.txt"));
    }
}

// The example displays the following output:
//     Unhandled Exception: System.ArgumentException: The path is not of a
legal form.
//         at System.IO.Path.NewNormalizePathLimitedChecks(String path, Int32
maxPathLength, Boolean expandShortPaths)
//         at System.IO.Path.NormalizePath(String path, Boolean fullCheck,
Int32 maxPathLength, Boolean expandShortPaths)
//         at System.IO.Path.InternalGetDirectoryName(String path)
//         at Example.Main()
```

To restore the method's previous behavior and prevent the exception, you can add the `Switch.System.IO.UseLegacyPathHandling` switch to the application configuration file for the example:

XML

```
<configuration>
    <runtime>
        <AppContextSwitchOverrides
value="Switch.System.IO.UseLegacyPathHandling=true" />
    </runtime>
</configuration>
```

See also

- [AppContext switch](#)



Collaborate with us on
GitHub



.NET feedback

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET is an open source project.
Select a link to provide feedback:

-  [Open a documentation issue](#)
-  [Provide product feedback](#)

Console apps in .NET

Article • 09/15/2021

.NET applications can use the [System.Console](#) class to read characters from and write characters to the console. Data from the console is read from the standard input stream, data to the console is written to the standard output stream, and error data to the console is written to the standard error output stream. These streams are automatically associated with the console when the application starts and are presented as the [In](#), [Out](#), and [Error](#) properties, respectively.

The value of the [Console.In](#) property is a [System.IO.TextReader](#) object, whereas the values of the [Console.Out](#) and [Console.Error](#) properties are [System.IO.TextWriter](#) objects. You can associate these properties with streams that do not represent the console, making it possible for you to point the stream to a different location for input or output. For example, you can redirect the output to a file by setting the [Console.Out](#) property to a [System.IO.StreamWriter](#), which encapsulates a [System.IO.FileStream](#) by means of the [Console.SetOut](#) method. The [Console.In](#) and [Console.Out](#) properties do not need to refer to the same stream.

Note

For more information about building console applications, including examples in C#, Visual Basic, and C++, see the documentation for the [Console](#) class.

If the console does not exist, for example, in a Windows Forms application, output written to the standard output stream will not be visible, because there is no console to write the information to. Writing information to an inaccessible console does not cause an exception to be raised. (You can always change the application type to [Console Application](#), for example, in the project property pages in Visual Studio).

The [System.Console](#) class has methods that can read individual characters or entire lines from the console. Other methods convert data and format strings, and then write the formatted strings to the console. For more information on formatting strings, see [Formatting types](#).

Tip

Console applications lack a message pump that starts by default. Therefore, console calls to Microsoft Win32 timers might fail.

See also

- [System.Console](#)
- [Formatting Types](#)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

System.Console class

Article • 01/08/2024

This article provides supplementary remarks to the reference documentation for this API.

The console is an operating system window where users interact with the operating system or with a text-based console application by entering text input through the computer keyboard, and by reading text output from the computer terminal. For example, in the Windows operating system, the console is called the Command Prompt window and accepts MS-DOS commands. The [Console](#) class provides basic support for applications that read characters from, and write characters to, the console.

Console I/O streams

When a console application starts, the operating system automatically associates three I/O streams with the console: standard input stream, standard output stream, and standard error output stream. Your application can read user input from the standard input stream; write normal data to the standard output stream; and write error data to the standard error output stream. These streams are presented to your application as the values of the [Console.In](#), [Console.Out](#), and [Console.Error](#) properties.

By default, the value of the [In](#) property is a [System.IO.TextReader](#) object that represents the keyboard, and the values of the [Out](#) and [Error](#) properties are [System.IO.TextWriter](#) objects that represent a console window. However, you can set these properties to streams that do not represent the console window or keyboard; for example, you can set these properties to streams that represent files. To redirect the standard input, standard output, or standard error stream, call the [Console.SetIn](#), [Console.SetOut](#), or [Console.SetError](#) method, respectively. I/O operations that use these streams are synchronized, which means that multiple threads can read from, or write to, the streams. This means that methods that are ordinarily asynchronous, such as [TextReader.ReadLineAsync](#), execute synchronously if the object represents a console stream.

Note

Do not use the [Console](#) class to display output in unattended applications, such as server applications. Calls to methods such as [Console.Write](#) and [Console.WriteLine](#) have no effect in GUI applications.

`Console` class members that work normally when the underlying stream is directed to a console might throw an exception if the stream is redirected, for example, to a file. Program your application to catch `System.IO.IOException` exceptions if you redirect a standard stream. You can also use the `IsOutputRedirected`, `IsInputRedirected`, and `IsErrorRedirected` properties to determine whether a standard stream is redirected before performing an operation that throws an `System.IO.IOException` exception.

It is sometimes useful to explicitly call the members of the stream objects represented by the `In`, `Out`, and `Error` properties. For example, by default, the `Console.ReadLine` method reads input from the standard input stream. Similarly, the `Console.WriteLine` method writes data to the standard output stream, and the data is followed by the default line termination string, which can be found at `Environment.NewLine`. However, the `Console` class does not provide a corresponding method to write data to the standard error output stream, or a property to change the line termination string for data written to that stream.

You can solve this problem by setting the `TextWriter.NewLine` property of the `Out` or `Error` property to another line termination string. For example, the following C# statement sets the line termination string for the standard error output stream to two carriage return and line feed sequences:

```
Console.Error.NewLine = "\r\n\r\n";
```

You can then explicitly call the `WriteLine` method of the error output stream object, as in the following C# statement:

```
Console.Error.WriteLine();
```

Screen buffer and console window

Two closely related features of the console are the screen buffer and the console window. Text is actually read from or written to streams owned by the console, but appear to be read from or written to an area owned by the console called the screen buffer. The screen buffer is an attribute of the console, and is organized as a rectangular grid of rows and columns where each grid intersection, or character cell, can contain a character. Each character has its own foreground color, and each character cell has its own background color.

The screen buffer is viewed through a rectangular region called the console window. The console window is another attribute of the console; it is not the console itself, which is an operating system window. The console window is arranged in rows and columns, is less than or equal to the size of the screen buffer, and can be moved to view different

areas of the underlying screen buffer. If the screen buffer is larger than the console window, the console automatically displays scroll bars so the console window can be repositioned over the screen buffer area.

A cursor indicates the screen buffer position where text is currently read or written. The cursor can be hidden or made visible, and its height can be changed. If the cursor is visible, the console window position is moved automatically so the cursor is always in view.

The origin for character cell coordinates in the screen buffer is the upper left corner, and the positions of the cursor and the console window are measured relative to that origin. Use zero-based indexes to specify positions; that is, specify the topmost row as row 0, and the leftmost column as column 0. The maximum value for the row and column indexes is [Int16.MaxValue](#).

Unicode support for the console

In general, the console reads input and writes output by using the current console code page, which the system locale defines by default. A code page can handle only a subset of available Unicode characters, so if you try to display characters that are not mapped by a particular code page, the console won't be able to display all characters or represent them accurately. The following example illustrates this problem. It tries to display the characters of the Cyrillic alphabet from U+0410 to U+044F to the console. If you run the example on a system that uses console code page 437, each character is replaced by a question mark (?), because Cyrillic characters do not map to the characters in code page 437.

C#

```
using System;

public class Example3
{
    public static void Main()
    {
        // Create a Char array for the modern Cyrillic alphabet,
        // from U+0410 to U+044F.
        int nChars = 0x044F - 0x0410 + 1;
        char[] chars = new char[nChars];
        ushort codePoint = 0x0410;
        for (int ctr = 0; ctr < chars.Length; ctr++)
        {
            chars[ctr] = (char)codePoint;
            codePoint++;
        }
    }
}
```

In addition to supporting code pages, the [Console](#) class supports UTF-8 encoding with the [UTF8Encoding](#) class. Beginning with the .NET Framework 4.5, the [Console](#) class also supports UTF-16 encoding with the [UnicodeEncoding](#) class. To display Unicode characters to the console, you set the [OutputEncoding](#) property to either [UTF8Encoding](#) or [UnicodeEncoding](#).

Support for Unicode characters requires the encoder to recognize a particular Unicode character, and also requires a font that has the glyphs needed to render that character. To successfully display Unicode characters to the console, the console font must be set to a non-raster or TrueType font such as Consolas or Lucida Console. The following example shows how you can programmatically change the font from a raster font to Lucida Console.

C#

```
using System;
using System.Runtime.InteropServices;

public class Example2
{
    [DllImport("kernel32.dll", SetLastError = true)]
    static extern IntPtr GetStdHandle(int nStdHandle);

    [DllImport("kernel32.dll", CharSet = CharSet.Unicode, SetLastError =
true)]
    static extern bool GetCurrentConsoleFontEx(
        IntPtr consoleOutput,
        bool maximumWindow,
        ref CONSOLE_FONT_INFO_EX lpConsoleCurrentFontEx);

    [DllImport("kernel32.dll", SetLastError = true)]
    static extern bool SetCurrentConsoleFontEx(
        IntPtr consoleOutput,
        bool maximumWindow,
        ref CONSOLE_FONT_INFO_EX lpConsoleCurrentFontEx);
}
```

```

        IntPtr consoleOutput,
        bool maximumWindow,
        CONSOLE_FONT_INFO_EX consoleCurrentFontEx);

private const int STD_OUTPUT_HANDLE = -11;
private const int TMPF_TRUETYPE = 4;
private const int LF_FACESIZE = 32;
private static IntPtr INVALID_HANDLE_VALUE = new IntPtr(-1);

public static unsafe void Main()
{
    string fontName = "Lucida Console";
    IntPtr hnd = GetStdHandle(STD_OUTPUT_HANDLE);
    if (hnd != INVALID_HANDLE_VALUE)
    {
        CONSOLE_FONT_INFO_EX info = new CONSOLE_FONT_INFO_EX();
        info.cbSize = (uint)Marshal.SizeOf(info);
        bool tt = false;
        // First determine whether there's already a TrueType font.
        if (GetCurrentConsoleFontEx(hnd, false, ref info))
        {
            tt = (info.FontFamily & TMPF_TRUETYPE) == TMPF_TRUETYPE;
            if (tt)
            {
                Console.WriteLine("The console already is using a
TrueType font.");
                return;
            }
            // Set console font to Lucida Console.
            CONSOLE_FONT_INFO_EX newInfo = new CONSOLE_FONT_INFO_EX();
            newInfo.cbSize = (uint)Marshal.SizeOf(newInfo);
            newInfo.FontFamily = TMPF_TRUETYPE;
            IntPtr ptr = new IntPtr(newInfo.FaceName);
            Marshal.Copy(fontName.ToCharArray(), 0, ptr,
fontName.Length);
            // Get some settings from current font.
            newInfo.dwFontSize = new COORD(info.dwFontSize.X,
info.dwFontSize.Y);
            newInfo.FontWeight = info.FontWeight;
            SetCurrentConsoleFontEx(hnd, false, newInfo);
        }
    }
}

[StructLayout(LayoutKind.Sequential)]
internal struct COORD
{
    internal short X;
    internal short Y;

    internal COORD(short x, short y)
    {
        X = x;
        Y = y;
    }
}

```

```

    }

[StructLayout(LayoutKind.Sequential)]
internal unsafe struct CONSOLE_FONT_INFO_EX
{
    internal uint cbSize;
    internal uint nFont;
    internal COORD dwFontSize;
    internal int FontFamily;
    internal int FontWeight;
    internal fixed char FaceName[LF_FACESIZE];
}
}

```

However, TrueType fonts can display only a subset of glyphs. For example, the Lucida Console font displays only 643 of the approximately 64,000 available characters from U+0021 to U+FB02. To see which characters a particular font supports, open the **Fonts** applet in Control Panel, choose the **Find a character** option, and choose the font whose character set you'd like to examine in the **Font** list of the **Character Map** window.

Windows uses font linking to display glyphs that are not available in a particular font. For information about font linking to display additional character sets, see [Globalization Step-by-Step: Fonts](#). Linked fonts are defined in the HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\FontLink\SystemLink subkey of the registry. Each entry associated with this subkey corresponds to the name of a base font, and its value is a string array that defines the font files and the fonts that are linked to the base font. Each member of the array defines a linked font and takes the form *font-file-name,font-name*. The following example illustrates how you can programmatically define a linked font named SimSun found in a font file named simsun.ttc that displays Simplified Han characters.

C#

```

using Microsoft.Win32;
using System;

public class Example
{
    public static void Main()
    {
        string valueName = "Lucida Console";
        string newFont = "simsun.ttc,SimSun";
        string[] fonts = null;
        RegistryValueKind kind = 0;
        bool toAdd;

        RegistryKey key = Registry.LocalMachine.OpenSubKey(
            @"Software\Microsoft\Windows

```

```

NT\CurrentVersion\FontLink\SystemLink",
    true);
if (key == null) {
    Console.WriteLine("Font linking is not enabled.");
}
else {
    // Determine if the font is a base font.
    string[] names = key.GetValueNames();
    if (Array.Exists(names, s => s.Equals(valueName,
                                                StringComparison.OrdinalIgnoreCase)))
{
    // Get the value's type.
    kind = key.GetValueKind(valueName);

    // Type should be RegistryValueKind.MultiString, but we can't be
sure.
    switch (kind) {
        case RegistryValueKind.String:
            fonts = new string[] { (string) key.GetValue(valueName) };
            break;
        case RegistryValueKind.MultiString:
            fonts = (string[]) key.GetValue(valueName);
            break;
        case RegistryValueKind.None:
            // Do nothing.
            fonts = new string[] { };
            break;
    }
    // Determine whether SimSun is a linked font.
    if (Array.FindIndex(fonts, s =>s.IndexOf("SimSun",
                                                StringComparison.OrdinalIgnoreCase)
>=0) >= 0) {
        Console.WriteLine("Font is already linked.");
        toAdd = false;
    }
    else {
        // Font is not a linked font.
        toAdd = true;
    }
}
else {
    // Font is not a base font.
    toAdd = true;
    fonts = new string[] { };
}

if (toAdd) {
    Array.Resize(ref fonts, fonts.Length + 1);
    fonts[fonts.GetUpperBound(0)] = newFont;
    // Change REG_SZ to REG_MULTI_SZ.
    if (kind == RegistryValueKind.String)
        key.DeleteValue(valueName, false);

    key.SetValue(valueName, fonts, RegistryValueKind.MultiString);
    Console.WriteLine("SimSun added to the list of linked fonts.");
}

```

```
        }

        if (key != null) key.Close();
    }
}
```

Unicode support for the console has the following limitations:

- UTF-32 encoding is not supported. The only supported Unicode encodings are UTF-8 and UTF-16, which are represented by the [UTF8Encoding](#) and [UnicodeEncoding](#) classes, respectively.
- Bidirectional output is not supported.
- Display of characters outside the Basic Multilingual Plane (that is, of surrogate pairs) is not supported, even if they are defined in a linked font file.
- Display of characters in complex scripts is not supported.
- Combining character sequences (that is, characters that consist of a base character and one or more combining characters) are displayed as separate characters. To work around this limitation, you can normalize the string to be displayed by calling the [String.Normalize](#) method before sending output to the console. In the following example, a string that contains the combining character sequence U+0061 U+0308 is displayed to the console as two characters before the output string is normalized, and as a single character after the [String.Normalize](#) method is called.

```
C#  
  
using System;  
using System.IO;  
  
public class Example1  
{  
    public static void Main()  
    {  
        char[] chars = { '\u0061', '\u0308' };  
  
        string combining = new String(chars);  
        Console.WriteLine(combining);  
  
        combining = combining.Normalize();  
        Console.WriteLine(combining);  
    }  
}  
// The example displays the following output:
```

```
//      a"  
//      ä
```

Normalization is a viable solution only if the Unicode standard for the character includes a pre-composed form that corresponds to a particular combining character sequence.

- If a font provides a glyph for a code point in the private use area, that glyph will be displayed. However, because characters in the private use area are application-specific, this may not be the expected glyph.

The following example displays a range of Unicode characters to the console. The example accepts three command-line parameters: the start of the range to display, the end of the range to display, and whether to use the current console encoding (`false`) or UTF-16 encoding (`true`). It assumes that the console is using a TrueType font.

C#

```
using System;  
using System.IO;  
using System.Globalization;  
using System.Text;  
  
public static class DisplayChars  
{  
    private static void Main(string[] args)  
    {  
        uint rangeStart = 0;  
        uint rangeEnd = 0;  
        bool setOutputEncodingToUnicode = true;  
        // Get the current encoding so we can restore it.  
        Encoding originalOutputEncoding = Console.OutputEncoding;  
  
        try  
        {  
            switch(args.Length)  
            {  
                case 2:  
                    rangeStart = uint.Parse(args[0], NumberStyles.HexNumber);  
                    rangeEnd = uint.Parse(args[1], NumberStyles.HexNumber);  
                    setOutputEncodingToUnicode = true;  
                    break;  
                case 3:  
                    if (!uint.TryParse(args[0], NumberStyles.HexNumber, null,  
out rangeStart))  
                        throw new ArgumentException(String.Format("{0} is not a  
valid hexadecimal number.", args[0]));  
  
                    if (!uint.TryParse(args[1], NumberStyles.HexNumber, null, out  
rangeEnd))
```

```

        throw new ArgumentException(String.Format("{0} is not a
valid hexadecimal number.", args[1]));

        bool.TryParse(args[2], out setOutputEncodingToUnicode);
        break;
    default:
        Console.WriteLine("Usage: {0} <{1}> <{2}> [{3}]",
            Environment.GetCommandLineArgs()[0],
            "startingCodePointInHex",
            "endingCodePointInHex",
            "<setOutputEncodingToUnicode?{true|false,
default:false}>");
        return;
    }

    if (setOutputEncodingToUnicode) {
        // This won't work before .NET Framework 4.5.
        try {
            // Set encoding using endianness of this system.
            // We're interested in displaying individual Char objects, so
            // we don't want a Unicode BOM or exceptions to be thrown on
            // invalid Char values.
            Console.OutputEncoding = new UnicodeEncoding(!
BitConverter.IsLittleEndian, false);
            Console.WriteLine("\nOutput encoding set to UTF-16");
        }
        catch (IOException) {
            Console.OutputEncoding = new UTF8Encoding();
            Console.WriteLine("Output encoding set to UTF-8");
        }
    }
    else {
        Console.WriteLine("The console encoding is {0} (code page {1})",
            Console.OutputEncoding.EncodingName,
            Console.OutputEncoding.CodePage);
    }
    DisplayRange(rangeStart, rangeEnd);
}
catch (ArgumentException ex) {
    Console.WriteLine(ex.Message);
}
finally {
    // Restore console environment.
    Console.OutputEncoding = originalOutputEncoding;
}
}

public static void DisplayRange(uint start, uint end)
{
    const uint upperRange = 0x10FFFF;
    const uint surrogateStart = 0xD800;
    const uint surrogateEnd = 0xDFFF;

    if (end <= start) {
        uint t = start;

```

```

        start = end;
        end = t;
    }

    // Check whether the start or end range is outside of last plane.
    if (start > upperRange)
        throw new ArgumentException(String.Format("0x{0:X5} is outside the
upper range of Unicode code points (0x{1:X5})",
                                            start, upperRange));
    if (end > upperRange)
        throw new ArgumentException(String.Format("0x{0:X5} is outside the
upper range of Unicode code points (0x{1:X5})",
                                            end, upperRange));

    // Since we're using 21-bit code points, we can't use U+D800 to
    // U+DFFF.
    if ((start < surrogateStart & end > surrogateStart) || (start >=
surrogateStart & start <= surrogateEnd ))
        throw new ArgumentException(String.Format("0x{0:X5}-0x{1:X5}
includes the surrogate pair range 0x{2:X5}-0x{3:X5}",
                                            start, end,
surrogateStart, surrogateEnd));
    uint last = RoundUpToMultipleOf(0x10, end);
    uint first = RoundDownToMultipleOf(0x10, start);

    uint rows = (last - first) / 0x10;

    for (uint r = 0; r < rows; ++r) {
        // Display the row header.
        Console.Write("{0:x5} ", first + 0x10 * r);

        for (uint c = 0; c < 0x10; ++c) {
            uint cur = (first + 0x10 * r + c);
            if (cur < start) {
                Console.Write($" {(char)(0x20)} ");
            }
            else if (end < cur) {
                Console.Write($" {(char)(0x20)} ");
            }
            else {
                // the cast to int is safe, since we know that val <=
upperRange.
                String chars = Char.ConvertFromUtf32( (int) cur);
                // Display a space for code points that are not valid
characters.
                if (CharUnicodeInfo.GetUnicodeCategory(chars[0]) ==
UnicodeCategory.OtherNotAssigned)
                    Console.Write($" {(char)(0x20)} ");
                // Display a space for code points in the private use area.
                else if (CharUnicodeInfo.GetUnicodeCategory(chars[0]) ==
UnicodeCategory.PrivateUse)
                    Console.Write($" {(char)(0x20)} ");
                // Is surrogate pair a valid character?
                // Note that the console will interpret the high and low
            }
        }
    }
}

```



```
//      004e0  З  З  Й  й - Й  й  Ö  ö -- Ø  ø  Ø  ø - Õ  õ  Ý  ý  
//      004f0  ÿ  ý  ÿ  ý - Ӧ  Ӧ  Ӯ  Ӯ  Ӱ  Ӱ  Ӳ  Ӳ  Ӵ  Ӵ  Ӷ  Ӷ
```

Common operations

The [Console](#) class contains the following methods for reading console input and writing console output:

- The overloads of the [ReadKey](#) method read an individual character.
- The [ReadLine](#) method reads an entire line of input.
- The [Write](#) method overloads convert an instance of a value type, an array of characters, or a set of objects to a formatted or unformatted string, and then write that string to the console.
- A parallel set of [WriteLine](#) method overloads output the same string as the [Write](#) overloads but also add a line termination string.

The [Console](#) class also contains methods and properties to perform the following operations:

- Get or set the size of the screen buffer. The [BufferSize](#) and [BufferWidth](#) properties let you get or set the buffer height and width, respectively, and the [SetBufferSize](#) method lets you set the buffer size in a single method call.
- Get or set the size of the console window. The [WindowHeight](#) and [WindowWidth](#) properties let you get or set the window height and width, respectively, and the [SetWindowSize](#) method lets you set the window size in a single method call.
- Get or set the size of the cursor. The [CursorSize](#) property specifies the height of the cursor in a character cell.
- Get or set the position of the console window relative to the screen buffer. The [WindowTop](#) and [WindowLeft](#) properties let you get or set the top row and leftmost column of the screen buffer that appears in the console window, and the [SetWindowPosition](#) method lets you set these values in a single method call.
- Get or set the position of the cursor by getting or setting the [CursorTop](#) and [CursorLeft](#) properties, or set the position of the cursor by calling the [SetCursorPosition](#) method.
- Move or clear data in the screen buffer by calling the [MoveBufferArea](#) or [Clear](#) method.

- Get or set the foreground and background colors by using the [ForegroundColor](#) and [BackgroundColor](#) properties, or reset the background and foreground to their default colors by calling the [ResetColor](#) method.
- Play the sound of a beep through the console speaker by calling the [Beep](#) method.

.NET Core notes

In .NET Framework on the desktop, the [Console](#) class uses the encoding returned by `GetConsoleCP` and `GetConsoleOutputCP`, which typically is a code page encoding. For example code, on systems whose culture is English (United States), code page 437 is the encoding that is used by default. However, .NET Core may make only a limited subset of these encodings available. Where this is the case, [Encoding.UTF8](#) is used as the default encoding for the console.

If your app depends on specific code page encodings, you can still make them available by doing the following *before* you call any [Console](#) methods:

1. Retrieve the [EncodingProvider](#) object from the `CodePagesEncodingProvider.Instance` property.
2. Pass the [EncodingProvider](#) object to the [Encoding.RegisterProvider](#) method to make the additional encodings supported by the encoding provider available.

The [Console](#) class will then automatically use the default system encoding rather than UTF8, provided that you have registered the encoding provider before calling any [Console](#) output methods.

Examples

The following example demonstrates how to read data from, and write data to, the standard input and output streams. Note that these streams can be redirected by using the [SetIn](#) and [SetOut](#) methods.

C#

```
using System;

public class Example4
{
    public static void Main()
    {
        Console.Write("Hello ");
        Console.WriteLine("World!");
    }
}
```

```
        Console.Write("Enter your name: ");
        string name = Console.ReadLine();
        Console.Write("Good day, ");
        Console.Write(name);
        Console.WriteLine("!");
    }
}

// The example displays output similar to the following:
//      Hello World!
//      Enter your name: James
//      Good day, James!
```

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

.NET is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

System.Random class

Article • 01/08/2024

This article provides supplementary remarks to the reference documentation for this API.

The [Random](#) class represents a pseudo-random number generator, which is an algorithm that produces a sequence of numbers that meet certain statistical requirements for randomness.

Pseudo-random numbers are chosen with equal probability from a finite set of numbers. The chosen numbers are not completely random because a mathematical algorithm is used to select them, but they are sufficiently random for practical purposes. The implementation of the [Random](#) class is based on a modified version of Donald E. Knuth's subtractive random number generator algorithm. For more information, see D. E. Knuth. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*. Addison-Wesley, Reading, MA, third edition, 1997.

To generate a cryptographically secure random number, such as one that's suitable for creating a random password, use the [RNGCryptoServiceProvider](#) class or derive a class from [System.Security.Cryptography.RandomNumberGenerator](#).

Instantiate the random number generator

You instantiate the random number generator by providing a seed value (a starting value for the pseudo-random number generation algorithm) to a [Random](#) class constructor. You can supply the seed value either explicitly or implicitly:

- The [Random\(Int32\)](#) constructor uses an explicit seed value that you supply.
- The [Random\(\)](#) constructor uses the default seed value. This is the most common way of instantiating the random number generator.

In .NET Framework, the default seed value is time-dependent. In .NET Core, the default seed value is produced by the thread-static, pseudo-random number generator.

If the same seed is used for separate [Random](#) objects, they will generate the same series of random numbers. This can be useful for creating a test suite that processes random values, or for replaying games that derive their data from random numbers. However, note that [Random](#) objects in processes running under different versions of .NET Framework may return different series of random numbers even if they're instantiated with identical seed values.

To produce different sequences of random numbers, you can make the seed value time-dependent, thereby producing a different series with each new instance of [Random](#). The parameterized [Random\(Int32\)](#) constructor can take an [Int32](#) value based on the number of ticks in the current time, whereas the parameterless [Random\(\)](#) constructor uses the system clock to generate its seed value. However, on .NET Framework only, because the clock has finite resolution, using the parameterless constructor to create different [Random](#) objects in close succession creates random number generators that produce identical sequences of random numbers. The following example illustrates how two [Random](#) objects that are instantiated in close succession in a .NET Framework application generate an identical series of random numbers. On most Windows systems, [Random](#) objects created within 15 milliseconds of one another are likely to have identical seed values.

C#

```
byte[] bytes1 = new byte[100];
byte[] bytes2 = new byte[100];
Random rnd1 = new Random();
Random rnd2 = new Random();

rnd1.NextBytes(bytes1);
rnd2.NextBytes(bytes2);

Console.WriteLine("First Series:");
for (int ctr = bytes1.GetLowerBound(0);
    ctr <= bytes1.GetUpperBound(0);
    ctr++) {
    Console.Write("{0, 5}", bytes1[ctr]);
    if ((ctr + 1) % 10 == 0) Console.WriteLine();
}

Console.WriteLine();

Console.WriteLine("Second Series:");
for (int ctr = bytes2.GetLowerBound(0);
    ctr <= bytes2.GetUpperBound(0);
    ctr++) {
    Console.Write("{0, 5}", bytes2[ctr]);
    if ((ctr + 1) % 10 == 0) Console.WriteLine();
}

// The example displays output like the following:
//      First Series:
//      97  129  149  54   22  208  120  105  68  177
//      113  214   30  172   74  218  116  230   89   18
//      12   112  130  105  116  180  190  200  187  120
//      7   198  233  158   58   51   50  170   98   23
//      21     1  113   74  146  245   34  255   96   24
//      232  255   23     9  167  240  255   44  194   98
//      18   175  173  204  169  171  236   127  114   23
```

```

//      167  202  132   65  253   11  254   56  214  127
//      145  191  104  163  143    7  174  224  247   73
//      52    6  231  255    5  101   83  165  160  231
//
//      Second Series:
//      97  129  149   54   22  208  120  105   68  177
//     113  214   30  172   74  218  116  230   89   18
//      12  112  130  105  116  180  190  200  187  120
//      7   198  233  158   58   51   50  170   98   23
//     21    1  113   74  146  245   34  255   96   24
//    232  255   23    9  167  240  255   44  194  98
//     18  175  173  204  169  171  236  127  114   23
//    167  202  132   65  253   11  254   56  214  127
//    145  191  104  163  143    7  174  224  247   73
//      52    6  231  255    5  101   83  165  160  231

```

To avoid this problem, create a single [Random](#) object instead of multiple objects. Note that the `Random` class in .NET Core does not have this limitation.

Avoid multiple instantiations

On .NET Framework, initializing two random number generators in a tight loop or in rapid succession creates two random number generators that can produce identical sequences of random numbers. In most cases, this is not the developer's intent and can lead to performance issues, because instantiating and initializing a random number generator is a relatively expensive process.

Both to improve performance and to avoid inadvertently creating separate random number generators that generate identical numeric sequences, we recommend that you create one [Random](#) object to generate many random numbers over time, instead of creating new [Random](#) objects to generate one random number.

However, the [Random](#) class isn't thread safe. If you call [Random](#) methods from multiple threads, follow the guidelines discussed in the next section.

Thread safety

Instead of instantiating individual [Random](#) objects, we recommend that you create a single [Random](#) instance to generate all the random numbers needed by your app.

However, [Random](#) objects are not thread safe. If your app calls [Random](#) methods from multiple threads, you must use a synchronization object to ensure that only one thread can access the random number generator at a time. If you don't ensure that the [Random](#) object is accessed in a thread-safe way, calls to methods that return random numbers return 0.

The following example uses the C# [lock Statement](#), the F# [lock function](#) and the Visual Basic [SyncLock statement](#) to ensure that a single random number generator is accessed by 11 threads in a thread-safe manner. Each thread generates 2 million random numbers, counts the number of random numbers generated and calculates their sum, and then updates the totals for all threads when it finishes executing.

C#

```
using System;
using System.Threading;

public class Example13
{
    [ThreadStatic] static double previous = 0.0;
    [ThreadStatic] static int perThreadCtr = 0;
    [ThreadStatic] static double perThreadTotal = 0.0;
    static CancellationTokenSource source;
    static CountdownEvent countdown;
    static Object randLock, numericLock;
    static Random rand;
    double totalValue = 0.0;
    int totalCount = 0;

    public Example13()
    {
        rand = new Random();
        randLock = new Object();
        numericLock = new Object();
        countdown = new CountdownEvent(1);
        source = new CancellationTokenSource();
    }

    public static void Main()
    {
        Example13 ex = new Example13();
        Thread.CurrentThread.Name = "Main";
        ex.Execute();
    }

    private void Execute()
    {
        CancellationToken token = source.Token;

        for (int threads = 1; threads <= 10; threads++)
        {
            Thread newThread = new Thread(this.GetRandomNumbers);
            newThread.Name = threads.ToString();
            newThread.Start(token);
        }
        this.GetRandomNumbers(token);

        countdown.Signal();
        // Make sure all threads have finished.
    }
}
```

```

        countdown.Wait();
        source.Dispose();

        Console.WriteLine("\nTotal random numbers generated: {0:N0}",
totalCount);
        Console.WriteLine("Total sum of all random numbers: {0:N2}",
totalValue);
        Console.WriteLine("Random number mean: {0:N4}", totalValue /
totalCount);
    }

    private void GetRandomNumbers(Object o)
{
    CancellationToken token = (CancellationToken)o;
    double result = 0.0;
    countdown.AddCount(1);

    try
    {
        for (int ctr = 0; ctr < 2000000; ctr++)
        {
            // Make sure there's no corruption of Random.
            token.ThrowIfCancellationRequested();

            lock (randLock)
            {
                result = rand.NextDouble();
            }
            // Check for corruption of Random instance.
            if ((result == previous) && result == 0)
            {
                source.Cancel();
            }
            else
            {
                previous = result;
            }
            perThreadCtr++;
            perThreadTotal += result;
        }
    }

    Console.WriteLine("Thread {0} finished execution.",
Thread.CurrentThread.Name);
    Console.WriteLine("Random numbers generated: {0:N0}",
perThreadCtr);
    Console.WriteLine("Sum of random numbers: {0:N2}",
perThreadTotal);
    Console.WriteLine("Random number mean: {0:N4}\n",
perThreadTotal / perThreadCtr);

    // Update overall totals.
    lock (numericLock)
    {
        totalCount += perThreadCtr;
        totalValue += perThreadTotal;
    }
}

```

```
        }
    }
    catch (OperationCanceledException e)
    {
        Console.WriteLine("Corruption in Thread {1}", e.GetType().Name,
Thread.CurrentThread.Name);
    }
    finally
    {
        countdown.Signal();
    }
}
}

// The example displays output like the following:
//      Thread 6 finished execution.
//      Random numbers generated: 2,000,000
//      Sum of random numbers: 1,000,491.05
//      Random number mean: 0.5002
//
//      Thread 10 finished execution.
//      Random numbers generated: 2,000,000
//      Sum of random numbers: 999,329.64
//      Random number mean: 0.4997
//
//      Thread 4 finished execution.
//      Random numbers generated: 2,000,000
//      Sum of random numbers: 1,000,166.89
//      Random number mean: 0.5001
//
//      Thread 8 finished execution.
//      Random numbers generated: 2,000,000
//      Sum of random numbers: 999,628.37
//      Random number mean: 0.4998
//
//      Thread Main finished execution.
//      Random numbers generated: 2,000,000
//      Sum of random numbers: 999,920.89
//      Random number mean: 0.5000
//
//      Thread 3 finished execution.
//      Random numbers generated: 2,000,000
//      Sum of random numbers: 999,370.45
//      Random number mean: 0.4997
//
//      Thread 7 finished execution.
//      Random numbers generated: 2,000,000
//      Sum of random numbers: 999,330.92
//      Random number mean: 0.4997
//
//      Thread 9 finished execution.
//      Random numbers generated: 2,000,000
//      Sum of random numbers: 1,000,172.79
//      Random number mean: 0.5001
//
//      Thread 5 finished execution.
```

```

//      Random numbers generated: 2,000,000
//      Sum of random numbers: 1,000,079.43
//      Random number mean: 0.5000
//
//      Thread 1 finished execution.
//      Random numbers generated: 2,000,000
//      Sum of random numbers: 999,817.91
//      Random number mean: 0.4999
//
//      Thread 2 finished execution.
//      Random numbers generated: 2,000,000
//      Sum of random numbers: 999,930.63
//      Random number mean: 0.5000
//
//      Total random numbers generated: 22,000,000
//      Total sum of all random numbers: 10,998,238.98
//      Random number mean: 0.4999

```

The example ensures thread-safety in the following ways:

- The [ThreadStaticAttribute](#) attribute is used to define thread-local variables that track the total number of random numbers generated and their sum for each thread.
- A lock (the `lock` statement in C#, the `lock` function in F# and the `SyncLock` statement in Visual Basic) protects access to the variables for the total count and sum of all random numbers generated on all threads.
- A semaphore (the [CountdownEvent](#) object) is used to ensure that the main thread blocks until all other threads complete execution.
- The example checks whether the random number generator has become corrupted by determining whether two consecutive calls to random number generation methods return 0. If corruption is detected, the example uses the [CancellationTokenSource](#) object to signal that all threads should be canceled.
- Before generating each random number, each thread checks the state of the [CancellationToken](#) object. If cancellation is requested, the example calls the [CancellationToken.ThrowIfCancellationRequested](#) method to cancel the thread.

The following example is identical to the first, except that it uses a [Task](#) object and a lambda expression instead of [Thread](#) objects.

C#

```

using System;
using System.Collections.Generic;
using System.Threading;
using System.Threading.Tasks;

```

```
public class Example15
{
    static Object randLock, numericLock;
    static Random rand;
    static CancellationTokenSource source;
    double totalValue = 0.0;
    int totalCount = 0;

    public Example15()
    {
        rand = new Random();
        randLock = new Object();
        numericLock = new Object();
        source = new CancellationTokenSource();
    }

    public static async Task Main()
    {
        Example15 ex = new Example15();
        Thread.CurrentThread.Name = "Main";
        await ex.Execute();
    }

    private async Task Execute()
    {
        List<Task> tasks = new List<Task>();

        for (int ctr = 0; ctr <= 10; ctr++)
        {
            CancellationToken token = source.Token;
            int taskNo = ctr;
            tasks.Add(Task.Run(() =>
            {
                double previous = 0.0;
                int taskCtr = 0;
                double taskTotal = 0.0;
                double result = 0.0;

                for (int n = 0; n < 2000000; n++)
                {
                    // Make sure there's no corruption of Random.
                    token.ThrowIfCancellationRequested();

                    lock (randLock)
                    {
                        result = rand.NextDouble();
                    }
                    // Check for corruption of Random instance.
                    if ((result == previous) && result == 0)
                    {
                        source.Cancel();
                    }
                    else
                    {
                        previous = result;
                    }
                }
            }));
        }
    }
}
```

```

        }
        taskCtr++;
        taskTotal += result;
    }

    // Show result.
    Console.WriteLine("Task {0} finished execution.",
taskNo);
    Console.WriteLine("Random numbers generated: {0:N0}",
taskCtr);
    Console.WriteLine("Sum of random numbers: {0:N2}",
taskTotal);
    Console.WriteLine("Random number mean: {0:N4}\n",
taskTotal / taskCtr);

    // Update overall totals.
    lock (numericLock)
    {
        totalCount += taskCtr;
        totalValue += taskTotal;
    }
},
token));
}
try
{
    await Task.WhenAll(tasks.ToArray());
    Console.WriteLine("\nTotal random numbers generated: {0:N0}",
totalCount);
    Console.WriteLine("Total sum of all random numbers: {0:N2}",
totalValue);
    Console.WriteLine("Random number mean: {0:N4}", totalValue /
totalCount);
}
catch (AggregateException e)
{
    foreach (Exception inner in e.InnerExceptions)
    {
        TaskCanceledException canc = inner as TaskCanceledException;
        if (canc != null)
            Console.WriteLine("Task #{0} cancelled.", canc.Task.Id);
        else
            Console.WriteLine("Exception: {0}",
inner.GetType().Name);
    }
}
finally
{
    source.Dispose();
}
}
}

// The example displays output like the following:
//      Task 1 finished execution.
//      Random numbers generated: 2,000,000

```

```
//      Sum of random numbers: 1,000,502.47
//      Random number mean: 0.5003
//
//      Task 0 finished execution.
//      Random numbers generated: 2,000,000
//      Sum of random numbers: 1,000,445.63
//      Random number mean: 0.5002
//
//      Task 2 finished execution.
//      Random numbers generated: 2,000,000
//      Sum of random numbers: 1,000,556.04
//      Random number mean: 0.5003
//
//      Task 3 finished execution.
//      Random numbers generated: 2,000,000
//      Sum of random numbers: 1,000,178.87
//      Random number mean: 0.5001
//
//      Task 4 finished execution.
//      Random numbers generated: 2,000,000
//      Sum of random numbers: 999,819.17
//      Random number mean: 0.4999
//
//      Task 5 finished execution.
//      Random numbers generated: 2,000,000
//      Sum of random numbers: 1,000,190.58
//      Random number mean: 0.5001
//
//      Task 6 finished execution.
//      Random numbers generated: 2,000,000
//      Sum of random numbers: 999,720.21
//      Random number mean: 0.4999
//
//      Task 7 finished execution.
//      Random numbers generated: 2,000,000
//      Sum of random numbers: 999,000.96
//      Random number mean: 0.4995
//
//      Task 8 finished execution.
//      Random numbers generated: 2,000,000
//      Sum of random numbers: 999,499.33
//      Random number mean: 0.4997
//
//      Task 9 finished execution.
//      Random numbers generated: 2,000,000
//      Sum of random numbers: 1,000,193.25
//      Random number mean: 0.5001
//
//      Task 10 finished execution.
//      Random numbers generated: 2,000,000
//      Sum of random numbers: 999,960.82
//      Random number mean: 0.5000
//
//      Total random numbers generated: 22,000,000
```

```
//      Total sum of all random numbers: 11,000,067.33
//      Random number mean: 0.5000
```

It differs from the first example in the following ways:

- The variables to keep track of the number of random numbers generated and their sum in each task are local to the task, so there is no need to use the [ThreadStaticAttribute](#) attribute.
- The static [Task.WaitAll](#) method is used to ensure that the main thread doesn't complete before all tasks have finished. There is no need for the [CountdownEvent](#) object.
- The exception that results from task cancellation is surfaced in the [Task.WaitAll](#) method. In the previous example, it is handled by each thread.

Generate different types of random numbers

The random number generator provides methods that let you generate the following kinds of random numbers:

- A series of [Byte](#) values. You determine the number of byte values by passing an array initialized to the number of elements you want the method to return to the [NextBytes](#) method. The following example generates 20 bytes.

C#

```
Random rnd = new Random();
Byte[] bytes = new Byte[20];
rnd.NextBytes(bytes);
for (int ctr = 1; ctr <= bytes.Length; ctr++)
{
    Console.Write("{0,3} ", bytes[ctr - 1]);
    if (ctr % 10 == 0) Console.WriteLine();
}

// The example displays output like the following:
//      141    48    189    66    134    212    211    71    161    56
//      181    166   220   133     9    252    222    57    62    62
```

- A single integer. You can choose whether you want an integer from 0 to a maximum value ([Int32.MaxValue](#) - 1) by calling the [Next\(\)](#) method, an integer between 0 and a specific value by calling the [Next\(Int32\)](#) method, or an integer within a range of values by calling the [Next\(Int32, Int32\)](#) method. In the parameterized overloads, the specified maximum value is exclusive; that is, the actual maximum number generated is one less than the specified value.

The following example calls the [Next\(Int32, Int32\)](#) method to generate 10 random numbers between -10 and 10. Note that the second argument to the method specifies the exclusive upper bound of the range of random values returned by the method. In other words, the largest integer that the method can return is one less than this value.

C#

```
Random rnd = new Random();
for (int ctr = 0; ctr < 10; ctr++)
{
    Console.WriteLine("{0,3} ", rnd.Next(-10, 11));
}

// The example displays output like the following:
//    2     9    -3     2     4    -7    -3    -8    -8      5
```

- A single floating-point value from 0.0 to less than 1.0 by calling the [NextDouble](#) method. The exclusive upper bound of the random number returned by the method is 1, so its actual upper bound is 0.9999999999999978. The following example generates 10 random floating-point numbers.

C#

```
Random rnd = new Random();
for (int ctr = 0; ctr < 10; ctr++)
{
    Console.WriteLine("{0,-19:R} ", rnd.NextDouble());
    if ((ctr + 1) % 3 == 0) Console.WriteLine();
}

// The example displays output like the following:
//    0.7911680553998649    0.0903414949264105    0.79776258291572455
//    0.615568345233597    0.652644504165577    0.84023809378977776
//    0.099662564741290441   0.91341467383942321   0.96018602045261581
//    0.74772306473354022
```

ⓘ Important

The [Next\(Int32, Int32\)](#) method allows you to specify the range of the returned random number. However, the `maxValue` parameter, which specifies the upper range returned number, is an exclusive, not an inclusive, value. This means that the method call `Next(0, 100)` returns a value between 0 and 99, and not between 0 and 100.

You can also use the [Random](#) class for such tasks as generating [random Boolean values](#), generating [random floating-point values in a specified range](#), generating [Generate random 64-bit integers](#), and [retrieving a unique element from an array or collection](#).

Substitute your own algorithm

You can implement your own random number generator by inheriting from the [Random](#) class and supplying your random number generation algorithm. To supply your own algorithm, you must override the [Sample](#) method, which implements the random number generation algorithm. You should also override the [Next\(\)](#), [Next\(Int32, Int32\)](#), and [NextBytes](#) methods to ensure that they call your overridden [Sample](#) method. You don't have to override the [Next\(Int32\)](#) and [NextDouble](#) methods.

For an example that derives from the [Random](#) class and modifies its default pseudo-random number generator, see the [Sample](#) reference page.

Retrieve the same sequence of random values

Sometimes you want to generate the same sequence of random numbers in software test scenarios and in game playing. Testing with the same sequence of random numbers allows you to detect regressions and confirm bug fixes. Using the same sequence of random number in games allows you to replay previous games.

You can generate the same sequence of random numbers by providing the same seed value to the [Random\(Int32\)](#) constructor. The seed value provides a starting value for the pseudo-random number generation algorithm. The following example uses 100100 as an arbitrary seed value to instantiate the [Random](#) object, displays 20 random floating-point values, and persists the seed value. It then restores the seed value, instantiates a new random number generator, and displays the same 20 random floating-point values. Note that the example may produce different sequences of random numbers if run on different versions of .NET.

C#

```
using System;
using System.IO;

public class Example12
{
    public static void Main()
    {
        int seed = 100100;
        ShowRandomNumbers(seed);
        Console.WriteLine();
    }

    static void ShowRandomNumbers(int seed)
    {
        Random r = new Random(seed);
        for (int i = 0; i < 20; i++)
        {
            double d = r.NextDouble();
            Console.WriteLine(d);
        }
    }
}
```

```
PersistSeed(seed);

        DisplayNewRandomNumbers();
    }

private static void ShowRandomNumbers(int seed)
{
    Random rnd = new Random(seed);
    for (int ctr = 0; ctr <= 20; ctr++)
        Console.WriteLine(rnd.NextDouble());
}

private static void PersistSeed(int seed)
{
    FileStream fs = new FileStream(@".\seed.dat", FileMode.Create);
    BinaryWriter bin = new BinaryWriter(fs);
    bin.Write(seed);
    bin.Close();
}

private static void DisplayNewRandomNumbers()
{
    FileStream fs = new FileStream(@".\seed.dat", FileMode.Open);
    BinaryReader bin = new BinaryReader(fs);
    int seed = bin.ReadInt32();
    bin.Close();

    Random rnd = new Random(seed);
    for (int ctr = 0; ctr <= 20; ctr++)
        Console.WriteLine(rnd.NextDouble());
}

// The example displays output like the following:
//      0.500193602172748
//      0.0209461245783354
//      0.465869495396442
//      0.195512794514891
//      0.928583675496552
//      0.729333720509584
//      0.381455668891527
//      0.0508996467343064
//      0.019261200921266
//      0.258578445417145
//      0.0177532266908107
//      0.983277184415272
//      0.483650274334313
//      0.0219647376900375
//      0.165910115077118
//      0.572085966622497
//      0.805291457942357
//      0.927985211335116
//      0.4228545699375
//      0.523320379910674
//      0.157783938645285
```

```
//          0.500193602172748
//          0.0209461245783354
//          0.465869495396442
//          0.195512794514891
//          0.928583675496552
//          0.729333720509584
//          0.381455668891527
//          0.0508996467343064
//          0.019261200921266
//          0.258578445417145
//          0.0177532266908107
//          0.983277184415272
//          0.483650274334313
//          0.0219647376900375
//          0.165910115077118
//          0.572085966622497
//          0.805291457942357
//          0.927985211335116
//          0.4228545699375
//          0.523320379910674
//          0.157783938645285
```

Retrieve unique sequences of random numbers

Providing different seed values to instances of the [Random](#) class causes each random number generator to produce a different sequence of values. You can provide a seed value either explicitly by calling the [Random\(Int32\)](#) constructor, or implicitly by calling the [Random\(\)](#) constructor. Most developers call the parameterless constructor, which uses the system clock. The following example uses this approach to instantiate two [Random](#) instances. Each instance displays a series of 10 random integers.

C#

```
using System;
using System.Threading;

public class Example16
{
    public static void Main()
    {
        Console.WriteLine("Instantiating two random number generators...");
        Random rnd1 = new Random();
        Thread.Sleep(2000);
        Random rnd2 = new Random();

        Console.WriteLine("\nThe first random number generator:");
        for (int ctr = 1; ctr <= 10; ctr++)
            Console.WriteLine("    {0}", rnd1.Next());
```

```

        Console.WriteLine("\nThe second random number generator:");
        for (int ctr = 1; ctr <= 10; ctr++)
            Console.WriteLine("    {0}", rnd2.Next());
    }
}

// The example displays output like the following:
//     Instantiating two random number generators...
//
//         The first random number generator:
//
//             643164361
//
//             1606571630
//
//             1725607587
//
//             2138048432
//
//             496874898
//
//             1969147632
//
//             2034533749
//
//             1840964542
//
//             412380298
//
//             47518930
//
//         The second random number generator:
//
//             1251659083
//
//             1514185439
//
//             1465798544
//
//             517841554
//
//             1821920222
//
//             195154223
//
//             1538948391
//
//             1548375095
//
//             546062716
//
//             897797880

```

However, because of its finite resolution, the system clock doesn't detect time differences that are less than approximately 15 milliseconds. Therefore, if your code calls the [Random\(\)](#) overload on .NET Framework to instantiate two [Random](#) objects in succession, you might inadvertently be providing the objects with identical seed values. (The [Random](#) class in .NET Core does not have this limitation.) To see this in the previous example, comment out the [Thread.Sleep](#) method call, and compile and run the example again.

To prevent this from happening, we recommend that you instantiate a single [Random](#) object rather than multiple ones. However, since [Random](#) isn't thread safe, you must use some synchronization device if you access a [Random](#) instance from multiple threads; for more information, see the [Thread safety](#) section. Alternately, you can use a delay mechanism, such as the [Sleep](#) method used in the previous example, to ensure that the instantiations occur more than 15 millisecond apart.

Retrieve integers in a specified range

You can retrieve integers in a specified range by calling the [Next\(Int32, Int32\)](#) method, which lets you specify both the lower and the upper bound of the numbers you'd like the random number generator to return. The upper bound is an exclusive, not an inclusive, value. That is, it isn't included in the range of values returned by the method. The following example uses this method to generate random integers between -10 and 10. Note that it specifies 11, which is one greater than the desired value, as the value of the `maxValue` argument in the method call.

C#

```
Random rnd = new Random();
for (int ctr = 1; ctr <= 15; ctr++)
{
    Console.WriteLine("{0,3}    ", rnd.Next(-10, 11));
    if (ctr % 5 == 0) Console.WriteLine();
}

// The example displays output like the following:
//      -2      -5      -1      -2      10
//      -3       6      -4      -8       3
//      -7      10       5      -2       4
```

Retrieve integers with a specified number of digits

You can call the [Next\(Int32, Int32\)](#) method to retrieve numbers with a specified number of digits. For example, to retrieve numbers with four digits (that is, numbers that range from 1000 to 9999), you call the [Next\(Int32, Int32\)](#) method with a `minValue` value of 1000 and a `maxValue` value of 10000, as the following example shows.

C#

```
Random rnd = new Random();
for (int ctr = 1; ctr <= 50; ctr++)
{
    Console.WriteLine("{0,3}    ", rnd.Next(1000, 10000));
    if (ctr % 10 == 0) Console.WriteLine();
}

// The example displays output like the following:
//      9570      8979      5770      1606      3818      4735      8495      7196      7070
//      2313
//      5279      6577      5104      5734      4227      3373      7376      6007      8193
//      5540
//      7558      3934      3819      7392      1113      7191      6947      4963      9179
//      7907
```

```
//      3391    6667    7269    1838    7317    1981    5154    7377    3297  
5320  
//      9869    8694    2684    4949    2999    3019    2357    5211    9604  
2593
```

Retrieve floating-point values in a specified range

The [NextDouble](#) method returns random floating-point values that range from 0 to less than 1. However, you'll often want to generate random values in some other range.

If the interval between the minimum and maximum desired values is 1, you can add the difference between the desired starting interval and 0 to the number returned by the [NextDouble](#) method. The following example does this to generate 10 random numbers between -1 and 0.

C#

```
Random rnd = new Random();  
for (int ctr = 1; ctr <= 10; ctr++)  
    Console.WriteLine(rnd.NextDouble() - 1);  
  
// The example displays output like the following:  
//      -0.930412760437658  
//      -0.164699016215605  
//      -0.9851692803135  
//      -0.43468508843085  
//      -0.177202483255976  
//      -0.776813320245972  
//      -0.0713201854710096  
//      -0.0912875561468711  
//      -0.540621722368813  
//      -0.232211863730201
```

To generate random floating-point numbers whose lower bound is 0 but upper bound is greater than 1 (or, in the case of negative numbers, whose lower bound is less than -1 and upper bound is 0), multiply the random number by the non-zero bound. The following example does this to generate 20 million random floating-point numbers that range from 0 to [Int64.MaxValue](#). It also displays the distribution of the random values generated by the method.

C#

```
const long ONE_TENTH = 922337203685477581;  
  
Random rnd = new Random();
```

```

double number;
int[] count = new int[10];

// Generate 20 million integer values between.
for (int ctr = 1; ctr <= 20000000; ctr++)
{
    number = rnd.NextDouble() * Int64.MaxValue;
    // Categorize random numbers into 10 groups.
    count[(int)(number / ONE_TENTH)]++;
}
// Display breakdown by range.
Console.WriteLine("{0,28} {1,32} {2,7}\n", "Range", "Count", "Pct.");
for (int ctr = 0; ctr <= 9; ctr++)
    Console.WriteLine("{0,25:N0}-{1,25:N0} {2,8:N0} {3,7:P2}", ctr *
ONE_TENTH,
                      ctr < 9 ? ctr * ONE_TENTH + ONE_TENTH - 1 :
Int64.MaxValue,
                      count[ctr], count[ctr] / 20000000.0);

// The example displays output like the following:
//          Range           Count
Pct.
//
//          0- 922,337,203,685,477,580 1,996,148   9.98
%
//      922,337,203,685,477,581-1,844,674,407,370,955,161 2,000,293   10.00
%
//      1,844,674,407,370,955,162-2,767,011,611,056,432,742 2,000,094   10.00
%
//      2,767,011,611,056,432,743-3,689,348,814,741,910,323 2,000,159   10.00
%
//      3,689,348,814,741,910,324-4,611,686,018,427,387,904 1,999,552   10.00
%
//      4,611,686,018,427,387,905-5,534,023,222,112,865,485 1,998,248   9.99
%
//      5,534,023,222,112,865,486-6,456,360,425,798,343,066 2,000,696   10.00
%
//      6,456,360,425,798,343,067-7,378,697,629,483,820,647 2,001,637   10.01
%
//      7,378,697,629,483,820,648-8,301,034,833,169,298,228 2,002,870   10.01
%
//      8,301,034,833,169,298,229-9,223,372,036,854,775,807 2,000,303   10.00
%

```

To generate random floating-point numbers between two arbitrary values, like the [Next\(Int32, Int32\)](#) method does for integers, use the following formula:

C#

```
Random.NextDouble() * (maxValue - minValue) + minValue
```

The following example generates 1 million random numbers that range from 10.0 to 11.0, and displays their distribution.

C#

```
Random rnd = new Random();
int lowerBound = 10;
int upperBound = 11;
int[] range = new int[10];
for (int ctr = 1; ctr <= 1000000; ctr++)
{
    Double value = rnd.NextDouble() * (upperBound - lowerBound) +
lowerBound;
    range[(int)Math.Truncate((value - lowerBound) * 10)]++;
}

for (int ctr = 0; ctr <= 9; ctr++)
{
    Double lowerRange = 10 + ctr * .1;
    Console.WriteLine("{0:N1} to {1:N1}: {2,8:N0} ({3,7:P2})",
                      lowerRange, lowerRange + .1, range[ctr],
                      range[ctr] / 1000000.0);
}

// The example displays output like the following:
//      10.0 to 10.1:  99,929  ( 9.99 %)
//      10.1 to 10.2:  100,189  (10.02 %)
//      10.2 to 10.3:  99,384  ( 9.94 %)
//      10.3 to 10.4:  100,240  (10.02 %)
//      10.4 to 10.5:  99,397  ( 9.94 %)
//      10.5 to 10.6:  100,580  (10.06 %)
//      10.6 to 10.7:  100,293  (10.03 %)
//      10.7 to 10.8:  100,135  (10.01 %)
//      10.8 to 10.9:  99,905  ( 9.99 %)
//      10.9 to 11.0:  99,948  ( 9.99 %)
```

Generate random Boolean values

The [Random](#) class doesn't provide methods that generate [Boolean](#) values. However, you can define your own class or method to do that. The following example defines a class, [BooleanGenerator](#), with a single method, [NextBoolean](#). The [BooleanGenerator](#) class stores a [Random](#) object as a private variable. The [NextBoolean](#) method calls the [Random.Next\(Int32, Int32\)](#) method and passes the result to the [Convert.ToBoolean\(Int32\)](#) method. Note that 2 is used as the argument to specify the upper bound of the random number. Since this is an exclusive value, the method call returns either 0 or 1.

C#

```

using System;

public class Example1
{
    public static void Main()
    {
        // Instantiate the Boolean generator.
        BooleanGenerator boolGen = new BooleanGenerator();
        int totalTrue = 0, totalFalse = 0;

        // Generate 1,000 random Booleans, and keep a running total.
        for (int ctr = 0; ctr < 1000000; ctr++)
        {
            bool value = boolGen.NextBoolean();
            if (value)
                totalTrue++;
            else
                totalFalse++;
        }
        Console.WriteLine("Number of true values: {0,7:N0} ({1:P3})",
                          totalTrue,
                          ((double)totalTrue) / (totalTrue + totalFalse));
        Console.WriteLine("Number of false values: {0,7:N0} ({1:P3})",
                          totalFalse,
                          ((double)totalFalse) / (totalTrue + totalFalse));
    }
}

public class BooleanGenerator
{
    Random rnd;

    public BooleanGenerator()
    {
        rnd = new Random();
    }

    public bool NextBoolean()
    {
        return rnd.Next(0, 2) == 1;
    }
}
// The example displays output like the following:
//      Number of true values: 500,004 (50.000 %)
//      Number of false values: 499,996 (50.000 %)

```

Instead of creating a separate class to generate random `Boolean` values, the example could simply have defined a single method. In that case, however, the `Random` object should have been defined as a class-level variable to avoid instantiating a new `Random` instance in each method call. In Visual Basic, the `Random` instance can be defined as a

Static variable in the `NextBoolean` method. The following example provides an implementation.

C#

```
Random rnd = new Random();

int totalTrue = 0, totalFalse = 0;

// Generate 1,000,000 random Booleans, and keep a running total.
for (int ctr = 0; ctr < 1000000; ctr++)
{
    bool value = NextBoolean();
    if (value)
        totalTrue++;
    else
        totalFalse++;
}
Console.WriteLine("Number of true values: {0,7:N0} ({1:P3})",
                  totalTrue,
                  ((double)totalTrue) / (totalTrue + totalFalse));
Console.WriteLine("Number of false values: {0,7:N0} ({1:P3})",
                  totalFalse,
                  ((double)totalFalse) / (totalTrue + totalFalse));

bool NextBoolean()
{
    return rnd.Next(0, 2) == 1;

// The example displays output like the following:
//      Number of true values: 499,777 (49.978 %)
//      Number of false values: 500,223 (50.022 %)
```

Generate random 64-bit integers

The overloads of the `Next` method return 32-bit integers. However, in some cases, you might want to work with 64-bit integers. You can do this as follows:

1. Call the `NextDouble` method to retrieve a double-precision floating point value.
2. Multiply that value by `Int64.MaxValue`.

The following example uses this technique to generate 20 million random long integers and categorizes them in 10 equal groups. It then evaluates the distribution of the random numbers by counting the number in each group from 0 to `Int64.MaxValue`. As the output from the example shows, the numbers are distributed more or less equally through the range of a long integer.

C#

```
const long ONE_TENTH = 922337203685477581;

Random rnd = new Random();
long number;
int[] count = new int[10];

// Generate 20 million long integers.
for (int ctr = 1; ctr <= 20000000; ctr++)
{
    number = (long)(rnd.NextDouble() * Int64.MaxValue);
    // Categorize random numbers.
    count[(int)(number / ONE_TENTH)]++;
}

// Display breakdown by range.
Console.WriteLine("{0,28} {1,32} {2,7}\n", "Range", "Count", "Pct.");
for (int ctr = 0; ctr <= 9; ctr++)
    Console.WriteLine("{0,25:N0}-{1,25:N0} {2,8:N0} {3,7:P2}", ctr *
ONE_TENTH,
                      ctr < 9 ? ctr * ONE_TENTH + ONE_TENTH - 1 :
Int64.MaxValue,
                      count[ctr], count[ctr] / 20000000.0);

// The example displays output like the following:
//          Range           Count
Pct.
//
//          0- 922,337,203,685,477,580  1,996,148   9.98
%
//          922,337,203,685,477,581-1,844,674,407,370,955,161  2,000,293   10.00
%
//          1,844,674,407,370,955,162-2,767,011,611,056,432,742  2,000,094   10.00
%
//          2,767,011,611,056,432,743-3,689,348,814,741,910,323  2,000,159   10.00
%
//          3,689,348,814,741,910,324-4,611,686,018,427,387,904  1,999,552   10.00
%
//          4,611,686,018,427,387,905-5,534,023,222,112,865,485  1,998,248   9.99
%
//          5,534,023,222,112,865,486-6,456,360,425,798,343,066  2,000,696   10.00
%
//          6,456,360,425,798,343,067-7,378,697,629,483,820,647  2,001,637   10.01
%
//          7,378,697,629,483,820,648-8,301,034,833,169,298,228  2,002,870   10.01
%
//          8,301,034,833,169,298,229-9,223,372,036,854,775,807  2,000,303   10.00
```

An alternative technique that uses bit manipulation does not generate truly random numbers. This technique calls [Next\(\)](#) to generate two integers, left-shifts one by 32 bits, and ORs them together. This technique has two limitations:

1. Because bit 31 is the sign bit, the value in bit 31 of the resulting long integer is always 0. This can be addressed by generating a random 0 or 1, left-shifting it 31 bits, and ORing it with the original random long integer.
2. More seriously, because the probability that the value returned by `Next()` will be 0, there will be few if any random numbers in the range 0x0-0x00000000FFFFFFFFFF.

Retrieve bytes in a specified range

The overloads of the `Next` method allow you to specify the range of random numbers, but the `NextBytes` method does not. The following example implements a `NextBytes` method that lets you specify the range of the returned bytes. It defines a `Random2` class that derives from `Random` and overloads its `NextBytes` method.

C#

```
using System;

public class Example3
{
    public static void Main()
    {
        Random2 rnd = new Random2();
        Byte[] bytes = new Byte[10000];
        int[] total = new int[101];
        rnd.NextBytes(bytes, 0, 101);

        // Calculate how many of each value we have.
        foreach (var value in bytes)
            total[value]++;

        // Display the results.
        for (int ctr = 0; ctr < total.Length; ctr++)
        {
            Console.Write("{0,3}: {1,-3}    ", ctr, total[ctr]);
            if ((ctr + 1) % 5 == 0) Console.WriteLine();
        }
    }

    public class Random2 : Random
    {
        public Random2() : base()
        { }

        public Random2(int seed) : base(seed)
        { }

        public void NextBytes(Byte[] bytes, byte minValue, byte maxValue)
        {

```

```

        for (int ctr = bytes.GetLowerBound(0); ctr <=
bytes.GetUpperBound(0); ctr++)
            bytes[ctr] = (byte)Next(minValue, maxValue);
    }
}

// The example displays output like the following:
//      0: 115      1: 119      2: 92      3: 98      4: 92
//      5: 102      6: 103      7: 84      8: 93      9: 116
//     10: 91      11: 98      12: 106     13: 91      14: 92
//     15: 101      16: 100      17: 96      18: 97      19: 100
//     20: 101      21: 106      22: 112     23: 82      24: 85
//     25: 102      26: 107      27: 98      28: 106     29: 102
//     30: 109      31: 108      32: 94      33: 101     34: 107
//     35: 101      36: 86       37: 100     38: 101     39: 102
//     40: 113      41: 95       42: 96       43: 89      44: 99
//     45: 81       46: 89       47: 105     48: 100     49: 85
//     50: 103      51: 103      52: 93       53: 89      54: 91
//     55: 97       56: 105      57: 97       58: 110     59: 86
//     60: 116      61: 94       62: 117     63: 98      64: 110
//     65: 93       66: 102      67: 100     68: 105     69: 83
//     70: 81       71: 97       72: 85       73: 70      74: 98
//     75: 100      76: 110      77: 114     78: 83      79: 90
//     80: 96       81: 112      82: 102     83: 102     84: 99
//     85: 81       86: 100      87: 93       88: 99      89: 118
//     90: 95       91: 124      92: 108     93: 96      94: 104
//     95: 106      96: 99       97: 99       98: 92      99: 99
//    100: 108

```

The `NextBytes(Byte[], Byte, Byte)` method wraps a call to the `Next(Int32, Int32)` method and specifies the minimum value and one greater than the maximum value (in this case, 0 and 101) that we want returned in the byte array. Because we are sure that the integer values returned by the `Next` method are within the range of the `Byte` data type, we can safely cast them (in C# and F#) or convert them (in Visual Basic) from integers to bytes.

Retrieve an element from an array or collection at random

Random numbers often serve as indexes to retrieve values from arrays or collections. To retrieve a random index value, you can call the `Next(Int32, Int32)` method, and use the lower bound of the array as the value of its `minValue` argument and one greater than the upper bound of the array as the value of its `maxValue` argument. For a zero-based array, this is equivalent to its `Length` property, or one greater than the value returned by the `Array.GetUpperBound` method. The following example randomly retrieves the name of a city in the United States from an array of cities.

C#

```
String[] cities = { "Atlanta", "Boston", "Chicago", "Detroit",
                    "Fort Wayne", "Greensboro", "Honolulu", "Indianapolis",
                    "Jersey City", "Kansas City", "Los Angeles",
                    "Milwaukee", "New York", "Omaha", "Philadelphia",
                    "Raleigh", "San Francisco", "Tulsa", "Washington" };
Random rnd = new Random();
int index = rnd.Next(0, cities.Length);
Console.WriteLine("Today's city of the day: {0}",
                  cities[index]);

// The example displays output like the following:
// Today's city of the day: Honolulu
```

Retrieve a unique element from an array or collection

A random number generator can always return duplicate values. As the range of numbers becomes smaller or the number of values generated becomes larger, the probability of duplicates grows. If random values must be unique, more numbers are generated to compensate for duplicates, resulting in increasingly poor performance.

There are a number of techniques to handle this scenario. One common solution is to create an array or collection that contains the values to be retrieved, and a parallel array that contains random floating-point numbers. The second array is populated with random numbers at the time the first array is created, and the [Array.Sort\(Array, Array\)](#) method is used to sort the first array by using the values in the parallel array.

For example, if you're developing a Solitaire game, you want to ensure that each card is used only once. Instead of generating random numbers to retrieve a card and tracking whether that card has already been dealt, you can create a parallel array of random numbers that can be used to sort the deck. Once the deck is sorted, your app can maintain a pointer to indicate the index of the next card on the deck.

The following example illustrates this approach. It defines a `Card` class that represents a playing card and a `Dealer` class that deals a deck of shuffled cards. The `Dealer` class constructor populates two arrays: a `deck` array that has class scope and that represents all the cards in the deck; and a local `order` array that has the same number of elements as the `deck` array and is populated with randomly generated `Double` values. The [Array.Sort\(Array, Array\)](#) method is then called to sort the `deck` array based on the values in the `order` array.

C#

```
using System;

// A class that represents an individual card in a playing deck.
public class Card
{
    public Suit Suit;
    public FaceValue FaceValue;

    public override String ToString()
    {
        return String.Format("{0:F} of {1:F}", this.FaceValue, this.Suit);
    }
}

public enum Suit { Hearts, Diamonds, Spades, Clubs };

public enum FaceValue
{
    Ace = 1, Two, Three, Four, Five, Six,
    Seven, Eight, Nine, Ten, Jack, Queen,
    King
};

public class Dealer
{
    Random rnd;
    // A deck of cards, without Jokers.
    Card[] deck = new Card[52];
    // Parallel array for sorting cards.
    Double[] order = new Double[52];
    // A pointer to the next card to deal.
    int ptr = 0;
    // A flag to indicate the deck is used.
    bool mustReshuffle = false;

    public Dealer()
    {
        rnd = new Random();
        // Initialize the deck.
        int deckCtr = 0;
        foreach (var suit in Enum.GetValues(typeof(Suit)))
        {
            foreach (var faceValue in Enum.GetValues(typeof(FaceValue)))
            {
                Card card = new Card();
                card.Suit = (Suit)suit;
                card.FaceValue = (FaceValue)faceValue;
                deck[deckCtr] = card;
                deckCtr++;
            }
        }
    }
}
```

```

        for (int ctr = 0; ctr < order.Length; ctr++)
            order[ctr] = rnd.NextDouble();

        Array.Sort(order, deck);
    }

    public Card[] Deal(int numberToDeal)
    {
        if (mustReshuffle)
        {
            Console.WriteLine("There are no cards left in the deck");
            return null;
        }

        Card[] cardsDealt = new Card[numberToDeal];
        for (int ctr = 0; ctr < numberToDeal; ctr++)
        {
            cardsDealt[ctr] = deck[ptr];
            ptr++;
            if (ptr == deck.Length)
                mustReshuffle = true;

            if (mustReshuffle & ctr < numberToDeal - 1)
            {
                Console.WriteLine("Can only deal the {0} cards remaining on
the deck.",
                                   ctr + 1);
                return cardsDealt;
            }
        }
        return cardsDealt;
    }
}

public class Example17
{
    public static void Main()
    {
        Dealer dealer = new Dealer();
        ShowCards(dealer.Deal(20));
    }

    private static void ShowCards(Card[] cards)
    {
        foreach (var card in cards)
            if (card != null)
                Console.WriteLine("{0} of {1}", card.FaceValue, card.Suit);
    }
}

// The example displays output like the following:
//      Six of Diamonds
//      King of Clubs
//      Eight of Clubs
//      Seven of Clubs
//      Queen of Clubs

```

```
//      King of Hearts
//      Three of Spades
//      Ace of Clubs
//      Four of Hearts
//      Three of Diamonds
//      Nine of Diamonds
//      Two of Hearts
//      Ace of Hearts
//      Three of Hearts
//      Four of Spades
//      Eight of Hearts
//      Queen of Diamonds
//      Two of Clubs
//      Four of Diamonds
//      Jack of Hearts
```

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

.NET is an open source project.
Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

.NET dependency injection

Article • 12/15/2023

.NET supports the dependency injection (DI) software design pattern, which is a technique for achieving [Inversion of Control \(IoC\)](#) between classes and their dependencies. Dependency injection in .NET is a built-in part of the framework, along with configuration, logging, and the options pattern.

A *dependency* is an object that another object depends on. Examine the following `MessageWriter` class with a `Write` method that other classes depend on:

C#

```
public class MessageWriter
{
    public void Write(string message)
    {
        Console.WriteLine($"MessageWriter.Write(message: \"{message}\")");
    }
}
```

A class can create an instance of the `MessageWriter` class to make use of its `Write` method. In the following example, the `MessageWriter` class is a dependency of the `Worker` class:

C#

```
public class Worker : BackgroundService
{
    private readonly MessageWriter _messageWriter = new();

    protected override async Task ExecuteAsync(CancellationToken stoppingToken)
    {
        while (!stoppingToken.IsCancellationRequested)
        {
            _messageWriter.Write($"Worker running at:
{DateTimeOffset.Now}");
            await Task.Delay(1_000, stoppingToken);
        }
    }
}
```

The class creates and directly depends on the `MessageWriter` class. Hard-coded dependencies, such as in the previous example, are problematic and should be avoided

for the following reasons:

- To replace `MessageWriter` with a different implementation, the `Worker` class must be modified.
- If `MessageWriter` has dependencies, they must also be configured by the `Worker` class. In a large project with multiple classes depending on `MessageWriter`, the configuration code becomes scattered across the app.
- This implementation is difficult to unit test. The app should use a mock or stub `MessageWriter` class, which isn't possible with this approach.

Dependency injection addresses these problems through:

- The use of an interface or base class to abstract the dependency implementation.
- Registration of the dependency in a service container. .NET provides a built-in service container, `IServiceProvider`. Services are typically registered at the app's start-up and appended to an `IServiceCollection`. Once all services are added, you use `BuildServiceProvider` to create the service container.
- *Injection* of the service into the constructor of the class where it's used. The framework takes on the responsibility of creating an instance of the dependency and disposing of it when it's no longer needed.

As an example, the `IMessageWriter` interface defines the `Write` method:

C#

```
namespace DependencyInjection.Example;

public interface IMessageWriter
{
    void Write(string message);
}
```

This interface is implemented by a concrete type, `MessageWriter`:

C#

```
namespace DependencyInjection.Example;

public class MessageWriter : IMessageWriter
{
    public void Write(string message)
    {
        Console.WriteLine($"MessageWriter.Write(message: \"{message}\")");
    }
}
```

The sample code registers the `IMessageWriter` service with the concrete type `MessageWriter`. The `AddSingleton` method registers the service with a singleton lifetime, the lifetime of the app. [Service lifetimes](#) are described later in this article.

C#

```
using DependencyInjection.Example;

HostApplicationBuilder builder = Host.CreateApplicationBuilder(args);

builder.Services.AddHostedService<Worker>();
builder.Services.AddSingleton<IMessageWriter, MessageWriter>();

using IHost host = builder.Build();

host.Run();
```

In the preceding code, the sample app:

- Creates a host app builder instance.
- Configures the services by registering:
 - The `Worker` as a hosted service. For more information, see [Worker Services in .NET](#).
 - The `IMessageWriter` interface as a singleton service with a corresponding implementation of the `MessageWriter` class.
- Builds the host and runs it.

The host contains the dependency injection service provider. It also contains all the other relevant services required to automatically instantiate the `Worker` and provide the corresponding `IMessageWriter` implementation as an argument.

C#

```
namespace DependencyInjection.Example;

public sealed class Worker(IMessageWriter messageWriter) : BackgroundService
{
    protected override async Task ExecuteAsync(CancellationToken stoppingToken)
    {
        while (!stoppingToken.IsCancellationRequested)
        {
            messageWriter.Write($"Worker running at: {DateTimeOffset.Now}");
            await Task.Delay(1_000, stoppingToken);
        }
    }
}
```

```
    }  
}
```

By using the DI pattern, the worker service:

- Doesn't use the concrete type `MessageWriter`, only the `IMessageWriter` interface that implements it. That makes it easy to change the implementation that the worker service uses without modifying the worker service.
- Doesn't create an instance of `MessageWriter`. The instance is created by the DI container.

The implementation of the `IMessageWriter` interface can be improved by using the built-in logging API:

```
C#  
  
namespace DependencyInjection.Example;  
  
public class LoggingMessageWriter(  
    ILogger<LoggingMessageWriter> logger) : IMessageWriter  
{  
    public void Write(string message) =>  
        logger.LogInformation("Info: {Msg}", message);  
}
```

The updated `AddSingleton` method registers the new `IMessageWriter` implementation:

```
C#  
  
builder.Services.AddSingleton<IMessageWriter, LoggingMessageWriter>();
```

The `HostApplicationBuilder` (`builder`) type is part of the `Microsoft.Extensions.Hosting` NuGet package.

`LoggingMessageWriter` depends on `ILogger<TCategoriesName>`, which it requests in the constructor. `ILogger<TCategoriesName>` is a [framework-provided service](#).

It's not unusual to use dependency injection in a chained fashion. Each requested dependency in turn requests its own dependencies. The container resolves the dependencies in the graph and returns the fully resolved service. The collective set of dependencies that must be resolved is typically referred to as a *dependency tree*, *dependency graph*, or *object graph*.

The container resolves `ILogger<TCategoriesName>` by taking advantage of [\(generic\) open types](#), eliminating the need to register every [\(generic\) constructed type](#).

With dependency injection terminology, a service:

- Is typically an object that provides a service to other objects, such as the `IMessageWriter` service.
- Is not related to a web service, although the service may use a web service.

The framework provides a robust logging system. The `IMessageWriter` implementations shown in the preceding examples were written to demonstrate basic DI, not to implement logging. Most apps shouldn't need to write loggers. The following code demonstrates using the default logging, which only requires the `Worker` to be registered as a hosted service [AddHostedService](#):

C#

```
public class Worker : BackgroundService
{
    private readonly ILogger<Worker> _logger;

    public Worker(ILogger<Worker> logger) =>
        _logger = logger;

    protected override async Task ExecuteAsync(CancellationToken stoppingToken)
    {
        while (!stoppingToken.IsCancellationRequested)
        {
            _logger.LogInformation("Worker running at: {time}",
DateTimeOffset.Now);
            await Task.Delay(1_000, stoppingToken);
        }
    }
}
```

Using the preceding code, there is no need to update `Program.cs`, because logging is provided by the framework.

Multiple constructor discovery rules

When a type defines more than one constructor, the service provider has logic for determining which constructor to use. The constructor with the most parameters where the types are DI-resolvable is selected. Consider the following C# example service:

C#

```
public class ExampleService
{
    public ExampleService()
```

```
{  
}  
  
public ExampleService(ILogger<ExampleService> logger)  
{  
    // omitted for brevity  
}  
  
public ExampleService(FooService fooService, BarService barService)  
{  
    // omitted for brevity  
}  
}
```

In the preceding code, assume that logging has been added and is resolvable from the service provider but the `FooService` and `BarService` types are not. The constructor with the `ILogger<ExampleService>` parameter is used to resolve the `ExampleService` instance. Even though there's a constructor that defines more parameters, the `FooService` and `BarService` types are not DI-resolvable.

If there's ambiguity when discovering constructors, an exception is thrown. Consider the following C# example service:

C#

```
public class ExampleService  
{  
    public ExampleService()  
    {  
    }  
  
    public ExampleService(ILogger<ExampleService> logger)  
    {  
        // omitted for brevity  
    }  
  
    public ExampleService(IOptions<ExampleOptions> options)  
    {  
        // omitted for brevity  
    }  
}
```

⚠ Warning

The `ExampleService` code with ambiguous DI-resolvable type parameters would throw an exception. Do **not** do this—it's intended to show what is meant by "ambiguous DI-resolvable types".

In the preceding example, there are three constructors. The first constructor is parameterless and requires no services from the service provider. Assume that both logging and options have been added to the DI container and are DI-resolvable services. When the DI container attempts to resolve the `ExampleService` type, it will throw an exception, as the two constructors are ambiguous.

You can avoid ambiguity by defining a constructor that accepts both DI-resolvable types instead:

C#

```
public class ExampleService
{
    public ExampleService()
    {

    }

    public ExampleService(
        ILogger<ExampleService> logger,
        IOptions<ExampleOptions> options)
    {
        // omitted for brevity
    }
}
```

Register groups of services with extension methods

Microsoft Extensions uses a convention for registering a group of related services. The convention is to use a single `Add{GROUP_NAME}` extension method to register all of the services required by a framework feature. For example, the `AddOptions` extension method registers all of the services required for using options.

Framework-provided services

When using any of the available host or app builder patterns, defaults are applied and services are registered by the framework. Consider some of the most popular host and app builder patterns:

- `Host.CreateDefaultBuilder()`
- `Host.CreateApplicationBuilder()`
- `WebHost.CreateDefaultBuilder()`
- `WebApplication.CreateBuilder()`

- `WebAssemblyHostBuilder.CreateDefault`
- `MauiApp.CreateBuilder`

After creating a builder from any of these APIs, the `IServiceCollection` has services defined by the framework, depending on [how the host was configured](#). For apps based on the .NET templates, the framework could register hundreds of services.

The following table lists a small sample of these framework-registered services:

[] [Expand table](#)

| Service Type | Lifetime |
|---|-----------|
| <code>Microsoft.Extensions.DependencyInjection.IServiceScopeFactory</code> | Singleton |
| <code>IHostApplicationLifetime</code> | Singleton |
| <code>Microsoft.Extensions.Logging.ILogger<TCategoryName></code> | Singleton |
| <code>Microsoft.Extensions.Logging.ILoggerFactory</code> | Singleton |
| <code>Microsoft.Extensions.ObjectPool.ObjectPoolProvider</code> | Singleton |
| <code>Microsoft.Extensions.Options.IConfigureOptions<TOptions></code> | Transient |
| <code>Microsoft.Extensions.Options.IOptions<TOptions></code> | Singleton |
| <code>System.Diagnostics.DiagnosticListener</code> | Singleton |
| <code>System.Diagnostics.DiagnosticSource</code> | Singleton |

Service lifetimes

Services can be registered with one of the following lifetimes:

- Transient
- Scoped
- Singleton

The following sections describe each of the preceding lifetimes. Choose an appropriate lifetime for each registered service.

Transient

Transient lifetime services are created each time they're requested from the service container. This lifetime works best for lightweight, stateless services. Register transient

services with [AddTransient](#).

In apps that process requests, transient services are disposed at the end of the request.

Scoped

For web applications, a scoped lifetime indicates that services are created once per client request (connection). Register scoped services with [AddScoped](#).

In apps that process requests, scoped services are disposed at the end of the request.

When using Entity Framework Core, the [AddDbContext](#) extension method registers `DbContext` types with a scoped lifetime by default.

Note

Do **not** resolve a scoped service from a singleton and be careful not to do so indirectly, for example, through a transient service. It may cause the service to have incorrect state when processing subsequent requests. It's fine to:

- Resolve a singleton service from a scoped or transient service.
- Resolve a scoped service from another scoped or transient service.

By default, in the development environment, resolving a service from another service with a longer lifetime throws an exception. For more information, see [Scope validation](#).

Singleton

Singleton lifetime services are created either:

- The first time they're requested.
- By the developer, when providing an implementation instance directly to the container. This approach is rarely needed.

Every subsequent request of the service implementation from the dependency injection container uses the same instance. If the app requires singleton behavior, allow the service container to manage the service's lifetime. Don't implement the singleton design pattern and provide code to dispose of the singleton. Services should never be disposed by code that resolved the service from the container. If a type or factory is registered as a singleton, the container disposes the singleton automatically.

Register singleton services with [AddSingleton](#). Singleton services must be thread safe and are often used in stateless services.

In apps that process requests, singleton services are disposed when the [ServiceProvider](#) is disposed on application shutdown. Because memory is not released until the app is shut down, consider memory use with a singleton service.

Service registration methods

The framework provides service registration extension methods that are useful in specific scenarios:

[+] Expand table

| Method | Automatic object disposal | Multiple implementations | Pass args |
|--|---------------------------|--------------------------|-----------|
| Add{LIFETIME}<{SERVICE}, {IMPLEMENTATION}>() | Yes | Yes | No |
| Example: | | | |
| services.AddSingleton<IMyDep, MyDep>(); | | | |
| Add{LIFETIME}<{SERVICE}>(sp => new {IMPLEMENTATION}) | Yes | Yes | Yes |
| Examples: | | | |
| services.AddSingleton<IMyDep>(sp => new MyDep()); services.AddSingleton<IMyDep>(sp => new MyDep(99)); | | | |
| Add{LIFETIME}<{IMPLEMENTATION}>() | Yes | No | No |
| Example: | | | |
| services.AddSingleton<MyDep>(); | | | |
| AddSingleton<{SERVICE}>(new {IMPLEMENTATION}) | No | Yes | Yes |
| Examples: | | | |
| services.AddSingleton<IMyDep>(new MyDep()); services.AddSingleton<IMyDep>(new MyDep(99)); | | | |

| Method | Automatic object disposal | Multiple implementations | Pass args |
|------------------------------------|---------------------------------|-----------------------------|--------------|
| AddSingleton(new {IMPLEMENTATION}) | No | No | Yes |

Examples:

```
services.AddSingleton(new MyDep());
services.AddSingleton(new MyDep(99));
```

For more information on type disposal, see the [Disposal of services](#) section.

Registering a service with only an implementation type is equivalent to registering that service with the same implementation and service type. This is why multiple implementations of a service cannot be registered using the methods that don't take an explicit service type. These methods can register multiple *instances* of a service, but they will all have the same *implementation* type.

Any of the above service registration methods can be used to register multiple service instances of the same service type. In the following example, `AddSingleton` is called twice with `IMessageWriter` as the service type. The second call to `AddSingleton` overrides the previous one when resolved as `IMessageWriter` and adds to the previous one when multiple services are resolved via `IEnumerable<IMessageWriter>`. Services appear in the order they were registered when resolved via `IEnumerable<{SERVICE}>`.

C#

```
using ConsoleDI.IEnumerableExample;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;

HostApplicationBuilder builder = Host.CreateApplicationBuilder(args);

builder.Services.AddSingleton<IMessageWriter, ConsoleMessageWriter>();
builder.Services.AddSingleton<IMessageWriter, LoggingMessageWriter>();
builder.Services.AddSingleton<ExampleService>();

using IHost host = builder.Build();

_ = host.Services.GetService<ExampleService>();

await host.RunAsync();
```

The preceding sample source code registers two implementations of the `IMessageWriter`.

C#

```
using System.Diagnostics;

namespace ConsoleDI.IEnumerableExample;

public sealed class ExampleService
{
    public ExampleService(
        IMessageWriter messageWriter,
        IEnumerable<IMessageWriter> messageWriters)
    {
        Trace.Assert(messageWriter is LoggingMessageWriter);

        var dependencyArray = messageWriters.ToArray();
        Trace.Assert(dependencyArray[0] is ConsoleMessageWriter);
        Trace.Assert(dependencyArray[1] is LoggingMessageWriter);
    }
}
```

The `ExampleService` defines two constructor parameters; a single `IMessageWriter`, and an `IEnumerable<IMessageWriter>`. The single `IMessageWriter` is the last implementation to have been registered, whereas the `IEnumerable<IMessageWriter>` represents all registered implementations.

The framework also provides `TryAdd{LIFETIME}` extension methods, which register the service only if there isn't already an implementation registered.

In the following example, the call to `AddSingleton` registers `ConsoleMessageWriter` as an implementation for `IMessageWriter`. The call to `TryAddSingleton` has no effect because `IMessageWriter` already has a registered implementation:

C#

```
services.AddSingleton<IMessageWriter, ConsoleMessageWriter>();
services.TryAddSingleton<IMessageWriter, LoggingMessageWriter>();
```

The `TryAddSingleton` has no effect, as it was already added and the "try" will fail. The `ExampleService` would assert the following:

C#

```
public class ExampleService
{
    public ExampleService(
        IMessageWriter messageWriter,
        IEnumerable<IMessageWriter> messageWriters)
```

```
{  
    Trace.Assert(messageWriter is ConsoleMessageWriter);  
    Trace.Assert(messageWriters.Single() is ConsoleMessageWriter);  
}  
}
```

For more information, see:

- [TryAdd](#)
- [TryAddTransient](#)
- [TryAddScoped](#)
- [TryAddSingleton](#)

The [TryAddEnumerable\(ServiceDescriptor\)](#) methods register the service only if there isn't already an implementation of *the same type*. Multiple services are resolved via `IEnumerable<{SERVICE}>`. When registering services, add an instance if one of the same types hasn't already been added. Library authors use `TryAddEnumerable` to avoid registering multiple copies of an implementation in the container.

In the following example, the first call to `TryAddEnumerable` registers `MessageWriter` as an implementation for `IMessageWriter1`. The second call registers `MessageWriter` for `IMessageWriter2`. The third call has no effect because `IMessageWriter1` already has a registered implementation of `MessageWriter`:

C#

```
public interface IMessageWriter1 { }  
public interface IMessageWriter2 { }  
  
public class MessageWriter : IMessageWriter1, IMessageWriter2  
{  
}  
  
services.TryAddEnumerable(ServiceDescriptor.Singleton<IMessageWriter1,  
MessageWriter>());  
services.TryAddEnumerable(ServiceDescriptor.Singleton<IMessageWriter2,  
MessageWriter>());  
services.TryAddEnumerable(ServiceDescriptor.Singleton<IMessageWriter1,  
MessageWriter>());
```

Service registration is generally order-independent except when registering multiple implementations of the same type.

`IServiceCollection` is a collection of `ServiceDescriptor` objects. The following example shows how to register a service by creating and adding a `ServiceDescriptor`:

C#

```
string secretKey = Configuration["SecretKey"];
var descriptor = new ServiceDescriptor(
    typeof(IMessageWriter),
    _ => new DefaultMessageWriter(secretKey),
    ServiceLifetime.Transient);

services.Add(descriptor);
```

The built-in `Add{LIFETIME}` methods use the same approach. For example, see the [AddScoped source code](#).

Constructor injection behavior

Services can be resolved by using:

- [IServiceProvider](#)
- [ActivatorUtilities](#):
 - Creates objects that aren't registered in the container.
 - Used with some framework features.

Constructors can accept arguments that aren't provided by dependency injection, but the arguments must assign default values.

When services are resolved by `IServiceProvider` or `ActivatorUtilities`, constructor injection requires a *public* constructor.

When services are resolved by `ActivatorUtilities`, constructor injection requires that only one applicable constructor exists. Constructor overloads are supported, but only one overload can exist whose arguments can all be fulfilled by dependency injection.

Scope validation

When the app runs in the `Development` environment and calls [CreateApplicationBuilder](#) to build the host, the default service provider performs checks to verify that:

- Scoped services aren't resolved from the root service provider.
- Scoped services aren't injected into singletons.

The root service provider is created when [BuildServiceProvider](#) is called. The root service provider's lifetime corresponds to the app's lifetime when the provider starts with the app and is disposed when the app shuts down.

Scoped services are disposed by the container that created them. If a scoped service is created in the root container, the service's lifetime is effectively promoted to singleton because it's only disposed by the root container when the app shuts down. Validating service scopes catches these situations when `BuildServiceProvider` is called.

Scope scenarios

The `IServiceScopeFactory` is always registered as a singleton, but the `IServiceProvider` can vary based on the lifetime of the containing class. For example, if you resolve services from a scope, and any of those services take an `IServiceProvider`, it'll be a scoped instance.

To achieve scoping services within implementations of `IHostedService`, such as the `BackgroundService`, do *not* inject the service dependencies via constructor injection. Instead, inject `IServiceScopeFactory`, create a scope, then resolve dependencies from the scope to use the appropriate service lifetime.

C#

```
namespace WorkerScope.Example;

public sealed class Worker(
    ILogger<Worker> logger,
    IServiceProvider serviceScopeFactory)
    : BackgroundService
{
    protected override async Task ExecuteAsync(CancellationToken
stoppingToken)
    {
        while (!stoppingToken.IsCancellationRequested)
        {
            using (IServiceScope scope = serviceScopeFactory.CreateScope())
            {
                try
                {
                    logger.LogInformation(
                        "Starting scoped work, provider hash: {hash}.",
                        scope.ServiceProvider.GetHashCode());

                    var store =
scope.ServiceProvider.GetRequiredService<IObjectStore>();
                    var next = await store.GetNextAsync();
                    logger.LogInformation("{next}", next);

                    var processor =
scope.ServiceProvider.GetRequiredService<IObjectProcessor>();
                    await processor.ProcessAsync(next);
                    logger.LogInformation("Processing {name}.", next.Name);
                }
            }
        }
    }
}
```

```
        var relay =
scope.ServiceProvider.GetRequiredService<IObjectRelay>();
        await relay.RelayAsync(next);
        logger.LogInformation("Processed results have been
 relayed.");
    }

    var marked = await store.MarkAsync(next);
    logger.LogInformation("Marked as processed: {next}",
 marked);
}

finally
{
    logger.LogInformation(
        "Finished scoped work, provider hash: {hash}.{nl}",
        scope.ServiceProvider.GetHashCode(),
Environment.NewLine);
}
}
}
}
```

In the preceding code, while the app is running, the background service:

- Depends on the `IServiceScopeFactory`.
 - Creates an `IServiceScope` for resolving additional services.
 - Resolves scoped services for consumption.
 - Works on processing objects and then relaying them, and finally marks them as processed.

From the sample source code, you can see how implementations of [IHostedService](#) can benefit from scoped service lifetimes.

Keyed services

.NET also supports service registrations and lookups based on a key, meaning it's possible to register multiple services with a different key, and use this key for the lookup.

For example, consider the case where you have different implementations of the interface `IMessageWriter`: `MemoryMessageWriter` and `QueueMessageWriter`.

You can register these services using the overload of the service registration methods (seen earlier) that supports a key as a parameter:

C#

```
services.AddKeyedSingleton<IMessageWriter, MemoryMessageWriter>("memory");
services.AddKeyedSingleton<IMessageWriter, QueueMessageWriter>("queue");
```

The `key` isn't limited to `string`, it can be any `object` you want, as long as the type correctly implements `Equals`.

In the constructor of the class that uses `IMessageWriter`, you add the `FromKeyedServicesAttribute` to specify the key of the service to resolve:

C#

```
public class ExampleService
{
    public ExampleService(
        [FromKeyedServices("queue")] IMessageWriter writer)
    {
        // Omitted for brevity...
    }
}
```

See also

- [Use dependency injection in .NET](#)
- [Dependency injection guidelines](#)
- [Dependency injection in ASP.NET Core](#)
- [NDC Conference Patterns for DI app development ↗](#)
- [Explicit dependencies principle](#)
- [Inversion of control containers and the dependency injection pattern \(Martin Fowler\) ↗](#)
- DI bugs should be created in the [github.com/dotnet/extensions ↗](#) repo

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

.NET is an open source project.
Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

Tutorial: Use dependency injection in .NET

Article • 12/11/2023

This tutorial shows how to use [dependency injection \(DI\) in .NET](#). With *Microsoft.Extensions*, DI is managed by adding services and configuring them in an [IServiceCollection](#). The [IHost](#) interface exposes the [IServiceProvider](#) instance, which acts as a container of all the registered services.

In this tutorial, you learn how to:

- ✓ Create a .NET console app that uses dependency injection
- ✓ Build and configure a [Generic Host](#)
- ✓ Write several interfaces and corresponding implementations
- ✓ Use service lifetime and scoping for DI

Prerequisites

- [.NET Core 3.1 SDK](#) or later.
- Familiarity with creating new .NET applications and installing NuGet packages.

Create a new console application

Using either the [dotnet new](#) command or an IDE new project wizard, create a new .NET console application named **ConsoleDI.Example**. Add the [Microsoft.Extensions.Hosting](#) NuGet package to the project.

Your new console app project file should resemble the following:

```
XML

<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net8.0</TargetFramework>
    <Nullable>enable</Nullable>
    <ImplicitUsings>true</ImplicitUsings>
    <RootNamespace>ConsoleDI.Example</RootNamespace>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.Extensions.Hosting" Version="8.0.0" />
  </ItemGroup>
</Project>
```

```
/>  
  </ItemGroup>  
  
</Project>
```

ⓘ Important

In this example, the [Microsoft.Extensions.Hosting](#) NuGet package is required to build and run the app. Some metapackages might contain the `Microsoft.Extensions.Hosting` package, in which case an explicit package reference isn't required.

Add interfaces

In this sample app, you'll learn how dependency injection handles service lifetime. You'll create several interfaces that represent different service lifetimes. Add the following interfaces to the project root directory:

IReportServiceLifetime.cs

C#

```
using Microsoft.Extensions.DependencyInjection;  
  
namespace ConsoleDI.Example;  
  
public interface IReportServiceLifetime  
{  
    Guid Id { get; }  
  
    ServiceLifetime Lifetime { get; }  
}
```

The `IReportServiceLifetime` interface defines:

- A `Guid Id` property that represents the unique identifier of the service.
- A `ServiceLifetime` property that represents the service lifetime.

ExampleTransientService.cs

C#

```
using Microsoft.Extensions.DependencyInjection;  
  
namespace ConsoleDI.Example;
```

```
public interface IExampleTransientService : IReportServiceLifetime
{
    ServiceLifetime IReportServiceLifetime.Lifetime =>
    ServiceLifetime.Transient;
}
```

IExampleScopedService.cs

C#

```
using Microsoft.Extensions.DependencyInjection;

namespace ConsoleDI.Example;

public interface IExampleScopedService : IReportServiceLifetime
{
    ServiceLifetime IReportServiceLifetime.Lifetime =>
    ServiceLifetime.Scoped;
}
```

IExampleSingletonService.cs

C#

```
using Microsoft.Extensions.DependencyInjection;

namespace ConsoleDI.Example;

public interface IExampleSingletonService : IReportServiceLifetime
{
    ServiceLifetime IReportServiceLifetime.Lifetime =>
    ServiceLifetime.Singleton;
}
```

All of the subinterfaces of `IReportServiceLifetime` explicitly implement the `IReportServiceLifetime.Lifetime` with a default. For example, `IExampleTransientService` explicitly implements `IReportServiceLifetime.Lifetime` with the `ServiceLifetime.Transient` value.

Add default implementations

The example implementations all initialize their `Id` property with the result of `Guid.NewGuid()`. Add the following default implementation classes for the various services to the project root directory:

ExampleTransientService.cs

C#

```
namespace ConsoleDI.Example;

internal sealed class ExampleTransientService : IExampleTransientService
{
    Guid IReportServiceLifetime.Id { get; } = Guid.NewGuid();
}
```

ExampleScopedService.cs

C#

```
namespace ConsoleDI.Example;

internal sealed class ExampleScopedService : IExampleScopedService
{
    Guid IReportServiceLifetime.Id { get; } = Guid.NewGuid();
}
```

ExampleSingletonService.cs

C#

```
namespace ConsoleDI.Example;

internal sealed class ExampleSingletonService : IExampleSingletonService
{
    Guid IReportServiceLifetime.Id { get; } = Guid.NewGuid();
}
```

Each implementation is defined as `internal sealed` and implements its corresponding interface. For example, `ExampleSingletonService` implements `IExampleSingletonService`.

Add a service that requires DI

Add the following service lifetime reporter class, which acts as a service to the console app:

ServiceLifetimeReporter.cs

C#

```

namespace ConsoleDI.Example;

internal sealed class ServiceLifetimeReporter(
    IExampleTransientService transientService,
    IExampleScopedService scopedService,
    IExampleSingletonService singletonService)
{
    public void ReportServiceLifetimeDetails(string lifetimeDetails)
    {
        Console.WriteLine(lifetimeDetails);

        LogService(transientService, "Always different");
        LogService(scopedService, "Changes only with lifetime");
        LogService(singletonService, "Always the same");
    }

    private static void LogService<T>(T service, string message)
        where T : IReportServiceLifetime =>
        Console.WriteLine(
            $"    {typeof(T).Name}: {service.Id} ({message})");
}

```

The `ServiceLifetimeReporter` defines a constructor that requires each of the aforementioned service interfaces, that is, `IExampleTransientService`, `IExampleScopedService`, and `IExampleSingletonService`. The object exposes a single method that allows the consumer to report on the service with a given `lifetimeDetails` parameter. When invoked, the `ReportServiceLifetimeDetails` method logs each service's unique identifier with the service lifetime message. The log messages help to visualize the service lifetime.

Register services for DI

Update `Program.cs` with the following code:

C#

```

using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
using ConsoleDI.Example;

HostApplicationBuilder builder = Host.CreateApplicationBuilder(args);

builder.Services.AddTransient<IExampleTransientService,
ExampleTransientService>();
builder.Services.AddScoped<IExampleScopedService, ExampleScopedService>();
builder.Services.AddSingleton<IExampleSingletonService,
ExampleSingletonService>();
builder.Services.AddTransient<ServiceLifetimeReporter>();

```

```

using IHost host = builder.Build();

ExemplifyServiceLifetime(host.Services, "Lifetime 1");
ExemplifyServiceLifetime(host.Services, "Lifetime 2");

await host.RunAsync();

static void ExemplifyServiceLifetime(IServiceProvider hostProvider, string
lifetime)
{
    using IServiceScope serviceScope = hostProvider.CreateScope();
    IServiceProvider provider = serviceScope.ServiceProvider;
    ServiceLifetimeReporter logger =
provider.GetRequiredService<ServiceLifetimeReporter>();
    logger.ReportServiceLifetimeDetails(
        $"{lifetime}: Call 1 to
provider.GetRequiredService<ServiceLifetimeReporter>()");

    Console.WriteLine("...");

    logger = provider.GetRequiredService<ServiceLifetimeReporter>();
    logger.ReportServiceLifetimeDetails(
        $"{lifetime}: Call 2 to
provider.GetRequiredService<ServiceLifetimeReporter>()");

    Console.WriteLine();
}

```

Each `services.Add{LIFETIME}<{SERVICE}>` extension method adds (and potentially configures) services. We recommend that apps follow this convention. Place extension methods in the [Microsoft.Extensions.DependencyInjection](#) namespace to encapsulate groups of service registrations. Including the namespace portion

[Microsoft.Extensions.DependencyInjection](#) for DI extension methods also:

- Allows them to be displayed in [IntelliSense](#) without adding additional `using` blocks.
- Prevents excessive `using` statements in the `Program` or `Startup` classes where these extension methods are typically called.

The app:

- Creates an [IHostBuilder](#) instance with [host builder settings](#).
- Configures services and adds them with their corresponding service lifetime.
- Calls [Build\(\)](#) and assigns an instance of [IHost](#).
- Calls [ExemplifyScoping](#), passing in the [IHost.Services](#).

Conclusion

In this sample app, you created several interfaces and corresponding implementations. Each of these services is uniquely identified and paired with a [ServiceLifetime](#). The sample app demonstrates registering service implementations against an interface, and how to register pure classes without backing interfaces. The sample app then demonstrates how dependencies defined as constructor parameters are resolved at run time.

When you run the app, it displays output similar to the following:

C#

```
// Sample output:  
// Lifetime 1: Call 1 to  
provider.GetRequiredService<ServiceLifetimeReporter>()  
//      IExampleTransientService: d08a27fa-87d2-4a06-98d7-2773af886125  
(Always different)  
//      IExampleScopedService: 402c83c9-b4ed-4be1-b78c-86be1b1d908d (Changes  
only with lifetime)  
//      IExampleSingletonService: a61f1ff4-0b14-4508-bd41-21d852484a7b  
(Always the same)  
// ...  
// Lifetime 1: Call 2 to  
provider.GetRequiredService<ServiceLifetimeReporter>()  
//      IExampleTransientService: b43d68fb-2c7b-4a9b-8f02-fc507c164326  
(Always different)  
//      IExampleScopedService: 402c83c9-b4ed-4be1-b78c-86be1b1d908d (Changes  
only with lifetime)  
//      IExampleSingletonService: a61f1ff4-0b14-4508-bd41-21d852484a7b  
(Always the same)  
//  
// Lifetime 2: Call 1 to  
provider.GetRequiredService<ServiceLifetimeReporter>()  
//      IExampleTransientService: f3856b59-ab3f-4bbd-876f-7bab0013d392  
(Always different)  
//      IExampleScopedService: bba80089-1157-4041-936d-e96d81dd9d1c (Changes  
only with lifetime)  
//      IExampleSingletonService: a61f1ff4-0b14-4508-bd41-21d852484a7b  
(Always the same)  
// ...  
// Lifetime 2: Call 2 to  
provider.GetRequiredService<ServiceLifetimeReporter>()  
//      IExampleTransientService: a8015c6a-08cd-4799-9ec3-2f2af9cbbfd2  
(Always different)  
//      IExampleScopedService: bba80089-1157-4041-936d-e96d81dd9d1c (Changes  
only with lifetime)  
//      IExampleSingletonService: a61f1ff4-0b14-4508-bd41-21d852484a7b  
(Always the same)
```

From the app output, you can see that:

- Transient services are always different, a new instance is created with every retrieval of the service.
- Scoped services change only with a new scope, but are the same instance within a scope.
- Singleton services are always the same, a new instance is only created once.

See also

- [Dependency injection guidelines](#)
- [Dependency injection in ASP.NET Core](#)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

.NET is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

Dependency injection guidelines

Article • 06/23/2023

This article provides general guidelines and best practices for implementing dependency injection in .NET applications.

Design services for dependency injection

When designing services for dependency injection:

- Avoid stateful, static classes and members. Avoid creating global state by designing apps to use singleton services instead.
- Avoid direct instantiation of dependent classes within services. Direct instantiation couples the code to a particular implementation.
- Make services small, well-factored, and easily tested.

If a class has many injected dependencies, it might be a sign that the class has too many responsibilities and violates the [Single Responsibility Principle \(SRP\)](#). Attempt to refactor the class by moving some of its responsibilities into new classes.

Disposal of services

The container is responsible for cleanup of types it creates, and calls [Dispose](#) on [IDisposable](#) instances. Services resolved from the container should never be disposed by the developer. If a type or factory is registered as a singleton, the container disposes the singleton automatically.

In the following example, the services are created by the service container and disposed automatically:

```
C#  
  
namespace ConsoleDisposable.Example;  
  
public sealed class TransientDisposable : IDisposable  
{  
    public void Dispose() => Console.WriteLine($"  
{nameof(TransientDisposable)}.Dispose()");  
}
```

The preceding disposable is intended to have a transient lifetime.

C#

```
namespace ConsoleDisposable.Example;

public sealed class ScopedDisposable : IDisposable
{
    public void Dispose() => Console.WriteLine($""
{nameof(ScopedDisposable)}.Dispose()");
}
```

The preceding disposable is intended to have a scoped lifetime.

C#

```
namespace ConsoleDisposable.Example;

public sealed class SingletonDisposable : IDisposable
{
    public void Dispose() => Console.WriteLine($""
{nameof(SingletonDisposable)}.Dispose()");
}
```

The preceding disposable is intended to have a singleton lifetime.

C#

```
using ConsoleDisposable.Example;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;

HostApplicationBuilder builder = Host.CreateApplicationBuilder(args);
builder.Services.AddTransient<TransientDisposable>();
builder.Services.AddScoped<ScopedDisposable>();
builder.Services.AddSingleton<SingletonDisposable>();

using IHost host = builder.Build();

ExemplifyDisposableScoping(host.Services, "Scope 1");
Console.WriteLine();

ExemplifyDisposableScoping(host.Services, "Scope 2");
Console.WriteLine();

await host.RunAsync();

static void ExemplifyDisposableScoping(IServiceProvider services, string
scope)
{
    Console.WriteLine($"{scope}...");

    using IServiceScope serviceScope = services.CreateScope();
```

```
    IServiceProvider provider = serviceScope.ServiceProvider;

    _ = provider.GetRequiredService<TransientDisposable>();
    _ = provider.GetRequiredService<ScopedDisposable>();
    _ = provider.GetRequiredService<SingletonDisposable>();
}
```

The debug console shows the following sample output after running:

```
Console

Scope 1...
ScopedDisposable.Dispose()
TransientDisposable.Dispose()

Scope 2...
ScopedDisposable.Dispose()
TransientDisposable.Dispose()

info: Microsoft.Hosting.Lifetime[0]
      Application started. Press Ctrl+C to shut down.
info: Microsoft.Hosting.Lifetime[0]
      Hosting environment: Production
info: Microsoft.Hosting.Lifetime[0]
      Content root path: .\configuration\console-di-
disposable\bin\Debug\net5.0
info: Microsoft.Hosting.Lifetime[0]
      Application is shutting down...
SingletonDisposable.Dispose()
```

Services not created by the service container

Consider the following code:

```
C#

// Register example service in IServiceCollection
builder.Services.AddSingleton(new ExampleService());
```

In the preceding code:

- The `ExampleService` instance is **not** created by the service container.
- The framework does **not** dispose of the services automatically.
- The developer is responsible for disposing the services.

IDisposable guidance for transient and shared instances

Transient, limited lifetime

Scenario

The app requires an [IDisposable](#) instance with a transient lifetime for either of the following scenarios:

- The instance is resolved in the root scope (root container).
- The instance should be disposed before the scope ends.

Solution

Use the factory pattern to create an instance outside of the parent scope. In this situation, the app would generally have a `Create` method that calls the final type's constructor directly. If the final type has other dependencies, the factory can:

- Receive an [IServiceProvider](#) in its constructor.
- Use [ActivatorUtilities.CreateInstance](#) to instantiate the instance outside of the container, while using the container for its dependencies.

Shared instance, limited lifetime

Scenario

The app requires a shared [IDisposable](#) instance across multiple services, but the [IDisposable](#) instance should have a limited lifetime.

Solution

Register the instance with a scoped lifetime. Use [IServiceScopeFactory.CreateScope](#) to create a new [IServiceScope](#). Use the scope's [IServiceProvider](#) to get required services. Dispose the scope when it's no longer needed.

General [IDisposable](#) guidelines

- Don't register [IDisposable](#) instances with a transient lifetime. Use the factory pattern instead.
- Don't resolve [IDisposable](#) instances with a transient or scoped lifetime in the root scope. The only exception to this is if the app creates/recreates and disposes [IServiceProvider](#), but this isn't an ideal pattern.
- Receiving an [IDisposable](#) dependency via DI doesn't require that the receiver implement [IDisposable](#) itself. The receiver of the [IDisposable](#) dependency shouldn't call [Dispose](#) on that dependency.

- Use scopes to control the lifetimes of services. Scopes aren't hierarchical, and there's no special connection among scopes.

For more information on resource cleanup, see [Implement a Dispose method](#), or [Implement a DisposeAsync method](#). Additionally, consider the [Disposable transient services captured by container](#) scenario as it relates to resource cleanup.

Default service container replacement

The built-in service container is designed to serve the needs of the framework and most consumer apps. We recommend using the built-in container unless you need a specific feature that it doesn't support, such as:

- Property injection
- Injection based on name
- Child containers
- Custom lifetime management
- `Func<T>` support for lazy initialization
- Convention-based registration

The following third-party containers can be used with ASP.NET Core apps:

- [Autofac ↗](#)
- [Dryloc ↗](#)
- [Grace ↗](#)
- [LightInject ↗](#)
- [Lamar ↗](#)
- [Stashbox ↗](#)
- [Simple Injector ↗](#)

Thread safety

Create thread-safe singleton services. If a singleton service has a dependency on a transient service, the transient service may also require thread safety depending on how it's used by the singleton.

The factory method of a singleton service, such as the second argument to `AddSingleton<TService>(IServiceCollection, Func<IServiceProvider, TService>)`, doesn't need to be thread-safe. Like a type (`static`) constructor, it's guaranteed to be called only once by a single thread.

Recommendations

- `async/await` and `Task` based service resolution isn't supported. Because C# doesn't support asynchronous constructors, use asynchronous methods after synchronously resolving the service.
- Avoid storing data and configuration directly in the service container. For example, a user's shopping cart shouldn't typically be added to the service container. Configuration should use the options pattern. Similarly, avoid "data holder" objects that only exist to allow access to another object. It's better to request the actual item via DI.
- Avoid static access to services. For example, avoid capturing `IApplicationBuilder.ApplicationServices` as a static field or property for use elsewhere.
- Keep **DI factories** fast and synchronous.
- Avoid using the *service locator pattern*. For example, don't invoke `GetService` to obtain a service instance when you can use DI instead.
- Another service locator variation to avoid is injecting a factory that resolves dependencies at run time. Both of these practices mix **Inversion of Control** strategies.
- Avoid calls to `BuildServiceProvider` when configuring services. Calling `BuildServiceProvider` typically happens when the developer wants to resolve a service when registering another service. Instead, use an overload that includes the `IServiceProvider` for this reason.
- **Disposable transient services are captured** by the container for disposal. This can turn into a memory leak if resolved from the top-level container.
- Enable scope validation to make sure the app doesn't have singletons that capture scoped services. For more information, see [Scope validation](#).

Like all sets of recommendations, you may encounter situations where ignoring a recommendation is required. Exceptions are rare, mostly special cases within the framework itself.

DI is an *alternative* to static/global object access patterns. You may not be able to realize the benefits of DI if you mix it with static object access.

Example anti-patterns

In addition to the guidelines in this article, there are several anti-patterns *you should avoid*. Some of these anti-patterns are learnings from developing the runtimes themselves.

⚠ Warning

These are example anti-patterns, *do not* copy the code, *do not* use these patterns, and avoid these patterns at all costs.

Disposable transient services captured by container

When you register *Transient* services that implement [IDisposable](#), by default the DI container will hold onto these references, and not [Dispose\(\)](#) of them until the container is disposed when application stops if they were resolved from the container, or until the scope is disposed if they were resolved from a scope. This can turn into a memory leak if resolved from container level.

C#

```
static void TransientDisposablesWithoutDispose()
{
    var services = new ServiceCollection();
    services.AddTransient<ExampleDisposable>();
    ServiceProvider serviceProvider = services.BuildServiceProvider();

    for (int i = 0; i < 1000; ++ i)
    {
        _ = serviceProvider.GetRequiredService<ExampleDisposable>();
    }

    // serviceProvider.Dispose();
}
```

In the preceding anti-pattern, 1,000 `ExampleDisposable` objects are instantiated and rooted. They will not be disposed of until the `serviceProvider` instance is disposed.

For more information on debugging memory leaks, see [Debug a memory leak in .NET](#).

Async DI factories can cause deadlocks

The term "DI factories" refers to the overload methods that exist when calling `Add{LIFETIME}`. There are overloads accepting a `Func<IServiceProvider, T>` where `T` is the service being registered, and the parameter is named `implementationFactory`. The `implementationFactory` can be provided as a lambda expression, local function, or method. If the factory is asynchronous, and you use `Task<TResult>.Result`, this will cause a deadlock.

C#

```
static void DeadLockWithAsyncFactory()
{
    var services = new ServiceCollection();
    services.AddSingleton<Foo>(implementationFactory: provider =>
    {
        Bar bar = GetBarAsync(provider).Result;
        return new Foo(bar);
    });

    services.AddSingleton<Bar>();

    using ServiceProvider serviceProvider = services.BuildServiceProvider();
    _ = serviceProvider.GetRequiredService<Foo>();
}
```

In the preceding code, the `implementationFactory` is given a lambda expression where the body calls `Task<TResult>.Result` on a `Task<Bar>` returning method. This *causes a deadlock*. The `GetBarAsync` method simply emulates an asynchronous work operation with `Task.Delay`, and then calls `GetRequiredService<T>(IServiceProvider)`.

C#

```
static async Task<Bar> GetBarAsync(IServiceProvider serviceProvider)
{
    // Emulate asynchronous work operation
    await Task.Delay(1000);

    return serviceProvider.GetRequiredService<Bar>();
}
```

For more information on asynchronous guidance, see [Asynchronous programming: Important info and advice](#). For more information debugging deadlocks, see [Debug a deadlock in .NET](#).

When you're running this anti-pattern and the deadlock occurs, you can view the two threads waiting from Visual Studio's Parallel Stacks window. For more information, see [View threads and tasks in the Parallel Stacks window](#).

Captive dependency

The term "[captive dependency](#)" was coined by [Mark Seemann](#), and refers to the misconfiguration of service lifetimes, where a longer-lived service holds a shorter-lived service captive.

C#

```
static void CaptiveDependency()
{
    var services = new ServiceCollection();
    services.AddSingleton<Foo>();
    services.AddScoped<Bar>();

    using ServiceProvider serviceProvider = services.BuildServiceProvider();
    // Enable scope validation
    // using ServiceProvider serviceProvider =
    services.BuildServiceProvider(validateScopes: true);

    _ = serviceProvider.GetRequiredService<Foo>();
}
```

In the preceding code, `Foo` is registered as a singleton and `Bar` is scoped - which on the surface seems valid. However, consider the implementation of `Foo`.

C#

```
namespace DependencyInjection.AntiPatterns
{
    public class Foo
    {
        public Foo(Bar bar)
        {
        }
    }
}
```

The `Foo` object requires a `Bar` object, and since `Foo` is a singleton, and `Bar` is scoped - this is a misconfiguration. As is, `Foo` would only be instantiated once, and it would hold onto `Bar` for its lifetime, which is longer than the intended scoped lifetime of `Bar`. You should consider validating scopes, by passing `validateScopes: true` to the `BuildServiceProvider(IServiceCollection, Boolean)`. When you validate the scopes, you'd get an `InvalidOperationException` with a message similar to "Cannot consume scoped service 'Bar' from singleton 'Foo'.".

For more information, see [Scope validation](#).

Scoped service as singleton

When using scoped services, if you're not creating a scope or within an existing scope - the service becomes a singleton.

C#

```
static void ScopedServiceBecomesSingleton()
{
    var services = new ServiceCollection();
    services.AddScoped<Bar>();

    using ServiceProvider serviceProvider =
services.BuildServiceProvider(validateScopes: true);
    using (IServiceScope scope = serviceProvider.CreateScope())
    {
        // Correctly scoped resolution
        Bar correct = scope.ServiceProvider.GetRequiredService<Bar>();
    }

    // Not within a scope, becomes a singleton
    Bar avoid = serviceProvider.GetRequiredService<Bar>();
}
```

In the preceding code, `Bar` is retrieved within an `IServiceScope`, which is correct. The anti-pattern is the retrieval of `Bar` outside of the scope, and the variable is named `avoid` to show which example retrieval is incorrect.

See also

- [Dependency injection in .NET](#)
- [Tutorial: Use dependency injection in .NET](#)

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

Configuration in .NET

Article • 07/20/2023

Configuration in .NET is performed using one or more [configuration providers](#). Configuration providers read configuration data from key-value pairs using various configuration sources:

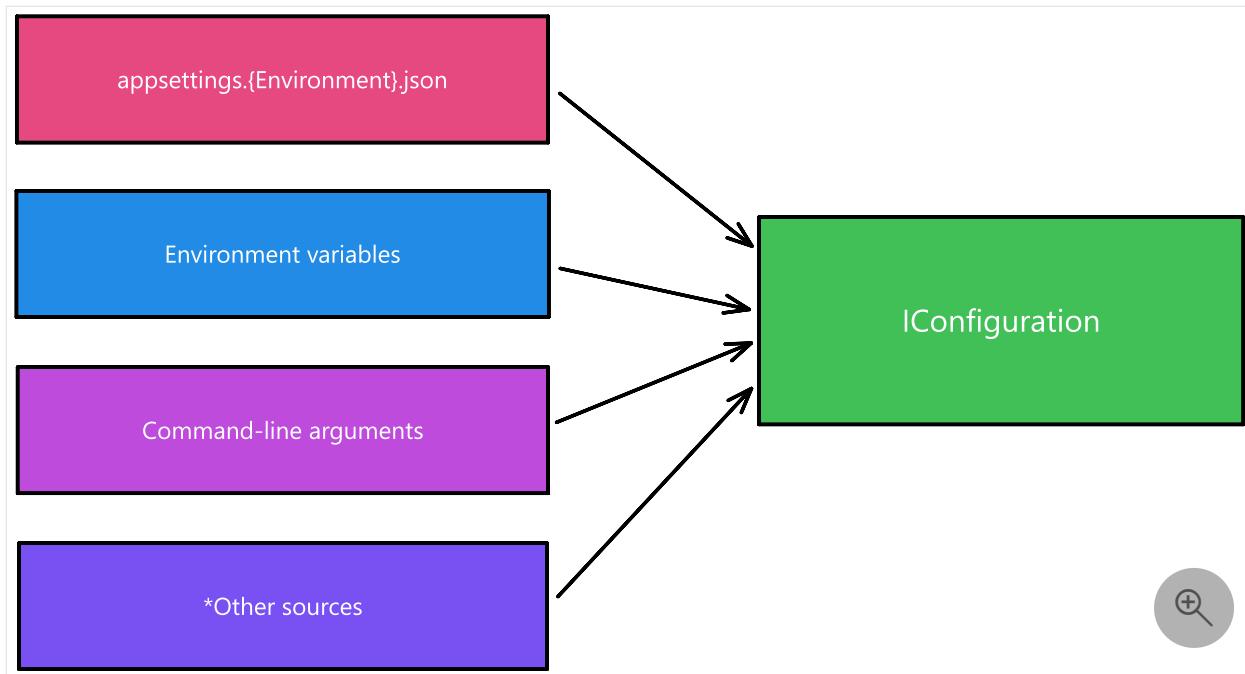
- Settings files, such as `appsettings.json`
- Environment variables
- [Azure Key Vault](#)
- [Azure App Configuration](#)
- Command-line arguments
- Custom providers, installed or created
- Directory files
- In-memory .NET objects
- Third-party providers

ⓘ Note

For information about configuring the .NET runtime itself, see [.NET Runtime configuration settings](#).

Concepts and abstractions

Given one or more configuration sources, the `IConfiguration` type provides a unified view of the configuration data. Configuration is read-only, and the configuration pattern isn't designed to be programmatically writable. The `IConfiguration` interface is a single representation of all the configuration sources, as shown in the following diagram:



Configure console apps

.NET console applications created using the [dotnet new](#) command template or Visual Studio by default *don't* expose configuration capabilities. To add configuration in a new .NET console application, [add a package reference to Microsoft.Extensions.Configuration](#). This package is the foundation for configuration in .NET apps. It provides the [ConfigurationBuilder](#) and related types.

C#

```

using Microsoft.Extensions.Configuration;

var configuration = new ConfigurationBuilder()
    .AddInMemoryCollection(new Dictionary<string, string?>()
{
    ["SomeKey"] = "SomeValue"
})
    .Build();

Console.WriteLine(configuration["SomeKey"]);

// Outputs:
// SomeValue

```

The preceding code:

- Creates a new [ConfigurationBuilder](#) instance.
- Adds an in-memory collection of key-value pairs to the configuration builder.
- Calls the [Build\(\)](#) method to create an [IConfiguration](#) instance.

- Writes the value of the `SomeKey` key to the console.

While this example uses an in-memory configuration, there are many configuration providers available, exposing functionality for file-based, environment variables, command line arguments, and other configuration sources. For more information, see [Configuration providers in .NET](#).

Alternative hosting approach

Commonly, your apps will do more than just read configuration. They'll likely use dependency injection, logging, and other services. The [.NET Generic Host](#) approach is recommended for apps that use these services. Instead, consider [adding a package reference](#) to [Microsoft.Extensions.Hosting](#). Modify the `Program.cs` file to match the following code:

C#

```
using Microsoft.Extensions.Hosting;

using IHost host = Host.CreateApplicationBuilder(args).Build();

// Application code should start here.

await host.RunAsync();
```

The `Host.CreateApplicationBuilder(String[])` method provides default configuration for the app in the following order, from highest to lowest priority:

1. Command-line arguments using the [Command-line configuration provider](#).
2. Environment variables using the [Environment Variables configuration provider](#).
3. [App secrets](#) when the app runs in the `Development` environment.
4. `appsettings.Environment.json` using the [JSON configuration provider](#). For example, `appsettings.Production.json` and `appsettings.Development.json`.
5. `appsettings.json` using the [JSON configuration provider](#).
6. [ChainedConfigurationProvider](#) : Adds an existing `IConfiguration` as a source.

Adding a configuration provider overrides previous configuration values. For example, the [Command-line configuration provider](#) overrides all values from other providers because it's added last. If `SomeKey` is set in both `appsettings.json` and the environment, the environment value is used because it was added after `appsettings.json`.

Binding

One of the key advantages of using the .NET configuration abstractions is the ability to bind configuration values to instances of .NET objects. For example, the JSON configuration provider can be used to map *appsettings.json* files to .NET objects and is used with [dependency injection](#). This enables the [options pattern](#), which uses classes to provide strongly typed access to groups of related settings. .NET configuration provides various abstractions. Consider the following interfaces:

- [IConfiguration](#): Represents a set of key/value application configuration properties.
- [IConfigurationRoot](#): Represents the root of an [IConfiguration](#) hierarchy.
- [IConfigurationSection](#): Represents a section of application configuration values.

These abstractions are agnostic to their underlying configuration provider ([IConfigurationProvider](#)). In other words, you can use an [IConfiguration](#) instance to access any configuration value from multiple providers.

The binder can use different approaches to process configuration values:

- Direct deserialization (using built-in converters) for primitive types.
- The [TypeConverter](#) for a complex type when the type has one.
- Reflection for a complex type that has properties.

① Note

The binder has a few limitations:

- Properties are ignored if they have private setters or their type can't be converted.
- Properties without corresponding configuration keys are ignored.

Binding hierarchies

Configuration values can contain hierarchical data. Hierarchical objects are represented with the use of the `:` delimiter in the configuration keys. To access a configuration value, use the `:` character to delimit a hierarchy. For example, consider the following configuration values:

JSON

```
{  
  "Parent": {  
    "FavoriteNumber": 7,  
    "Child": {  
      "Name": "Example",  
    }  
  }  
}
```

```

        "GrandChild": {
            "Age": 3
        }
    }
}

```

The following table represents example keys and their corresponding values for the preceding example JSON:

| Key | Value |
|-------------------------------|-----------|
| "Parent:FavoriteNumber" | 7 |
| "Parent:Child:Name" | "Example" |
| "Parent:Child:GrandChild:Age" | 3 |

Basic example

To access configuration values in their basic form, without the assistance of the *generic host* approach, use the [ConfigurationBuilder](#) type directly.

Tip

The [System.Configuration.ConfigurationBuilder](#) type is different to the [Microsoft.Extensions.Configuration.ConfigurationBuilder](#) type. All of this content is specific to the `Microsoft.Extensions.*` NuGet packages and namespaces.

Consider the following C# project:

XML

```

<Project Sdk="Microsoft.NET.Sdk">

    <PropertyGroup>
        <OutputType>Exe</OutputType>
        <TargetFramework>net7.0</TargetFramework>
        <Nullable>enable</Nullable>
        <ImplicitUsings>true</ImplicitUsings>
    </PropertyGroup>

    <ItemGroup>
        <Content Include="appsettings.json">
            <CopyToOutputDirectory>Always</CopyToOutputDirectory>
        </Content>
    </ItemGroup>

```

```

<ItemGroup>
    <PackageReference Include="Microsoft.Extensions.Configuration.Binder"
Version="8.0.0" />
    <PackageReference Include="Microsoft.Extensions.Configuration.Json"
Version="8.0.0" />
    <PackageReference
Include="Microsoft.Extensions.Configuration.EnvironmentVariables"
Version="8.0.0" />
</ItemGroup>

</Project>

```

The preceding project file references several configuration NuGet packages:

- [Microsoft.Extensions.Configuration.Binder](#): Functionality to bind an object to data in configuration providers for `Microsoft.Extensions.Configuration`.
- [Microsoft.Extensions.Configuration.Json](#): JSON configuration provider implementation for `Microsoft.Extensions.Configuration`.
- [Microsoft.Extensions.Configuration.EnvironmentVariables](#): Environment variables configuration provider implementation for `Microsoft.Extensions.Configuration`.

Consider an example `appsettings.json` file:

JSON

```
{
  "Settings": {
    "KeyOne": 1,
    "KeyTwo": true,
    "KeyThree": {
      "Message": "Oh, that's nice...",
      "SupportedVersions": {
        "v1": "1.0.0",
        "v3": "3.0.7"
      }
    },
    "IPAddressRange": [
      "46.36.198.121",
      "46.36.198.122",
      "46.36.198.123",
      "46.36.198.124",
      "46.36.198.125"
    ]
  }
}
```

Now, given this JSON file, here's an example consumption pattern using the configuration builder directly:

C#

```
using Microsoft.Extensions.Configuration;

// Build a config object, using env vars and JSON providers.
IConfigurationRoot config = new ConfigurationBuilder()
    .AddJsonFile("appsettings.json")
    .AddEnvironmentVariables()
    .Build();

// Get values from the config given their key and their target type.
Settings? settings = config.GetRequiredSection("Settings").Get<Settings>();

// Write the values to the console.
Console.WriteLine($"KeyOne = {settings?.KeyOne}");
Console.WriteLine($"KeyTwo = {settings?.KeyTwo}");
Console.WriteLine($"KeyThree:Message = {settings?.KeyThree?.Message}");

// Application code which might rely on the config could start here.

// This will output the following:
// KeyOne = 1
// KeyTwo = True
// KeyThree:Message = Oh, that's nice...
```

The preceding C# code:

- Instantiates a [ConfigurationBuilder](#).
- Adds the "appsettings.json" file to be recognized by the JSON configuration provider.
- Adds environment variables as being recognized by the Environment Variable configuration provider.
- Gets the required "Settings" section and the corresponding `Settings` instance by using the `config` instance.

The `Settings` object is shaped as follows:

C#

```
public sealed class Settings
{
    public required int KeyOne { get; set; }
    public required bool KeyTwo { get; set; }
    public required NestedSettings KeyThree { get; set; } = null!;
}
```

C#

```
public sealed class NestedSettings
{
    public required string Message { get; set; } = null!;
}
```

Basic example with hosting

To access the `IConfiguration` value, you can rely again on the [Microsoft.Extensions.Hosting](#) NuGet package. Create a new console application, and paste the following project file contents into it:

XML

```
<Project Sdk="Microsoft.NET.Sdk">

<PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net7.0</TargetFramework>
    <Nullable>enable</Nullable>
    <ImplicitUsings>true</ImplicitUsings>
</PropertyGroup>

<ItemGroup>
    <Content Include="appsettings.json">
        <CopyToOutputDirectory>Always</CopyToOutputDirectory>
    </Content>
</ItemGroup>

<ItemGroup>
    <PackageReference Include="Microsoft.Extensions.Hosting" Version="8.0.0" />
</ItemGroup>

</Project>
```

The preceding project file defines that:

- The application is an executable.
- An *appsettings.json* file is to be copied to the output directory when the project is compiled.
- The `Microsoft.Extensions.Hosting` NuGet package reference is added.

Add the *appsettings.json* file at the root of the project with the following contents:

JSON

```
{  
    "KeyOne": 1,  
    "KeyTwo": true,  
    "KeyThree": {  
        "Message": "Thanks for checking this out!"  
    }  
}
```

Replace the contents of the `Program.cs` file with the following C# code:

C#

```
using Microsoft.Extensions.Configuration;  
using Microsoft.Extensions.DependencyInjection;  
using Microsoft.Extensions.Hosting;  
  
using IHost host = Host.CreateApplicationBuilder(args).Build();  
  
// Ask the service provider for the configuration abstraction.  
IConfiguration config = host.Services.GetRequiredService< IConfiguration>();  
  
// Get values from the config given their key and their target type.  
int keyOneValue = config.GetValue< int >("KeyOne");  
bool keyTwoValue = config.GetValue< bool >("KeyTwo");  
string? keyThreeNestedValue = config.GetValue< string >("KeyThree:Message");  
  
// Write the values to the console.  
Console.WriteLine($"KeyOne = {keyOneValue}");  
Console.WriteLine($"KeyTwo = {keyTwoValue}");  
Console.WriteLine($"KeyThree:Message = {keyThreeNestedValue}");  
  
// Application code which might rely on the config could start here.  
  
await host.RunAsync();  
  
// This will output the following:  
// KeyOne = 1  
// KeyTwo = True  
// KeyThree:Message = Thanks for checking this out!
```

When you run this application, the `Host.CreateApplicationBuilder` defines the behavior to discover the JSON configuration and expose it through the `IConfiguration` instance. From the `host` instance, you can ask the service provider for the `IConfiguration` instance and then ask it for values.

 Tip

Using the raw `IConfiguration` instance in this way, while convenient, doesn't scale very well. When applications grow in complexity, and their corresponding configurations become more complex, we recommend that you use the ***options pattern*** as an alternative.

Basic example with hosting and using the indexer API

Consider the same `appsettings.json` file contents from the previous example:

JSON

```
{  
  "SupportedVersions": {  
    "v1": "1.0.0",  
    "v3": "3.0.7"  
  },  
  "IPAddressRange": [  
    "46.36.198.123",  
    "46.36.198.124",  
    "46.36.198.125"  
  ]  
}
```

Replace the contents of the `Program.cs` file with the following C# code:

C#

```
using Microsoft.Extensions.Configuration;  
using Microsoft.Extensions.DependencyInjection;  
using Microsoft.Extensions.Hosting;  
  
using IHost host = Host.CreateApplicationBuilder(args).Build();  
  
// Ask the service provider for the configuration abstraction.  
IConfiguration config = host.Services.GetRequiredService<IConfiguration>();  
  
// Get values from the config given their key and their target type.  
string? ipOne = config["IPAddressRange:0"];  
string? ipTwo = config["IPAddressRange:1"];  
string? ipThree = config["IPAddressRange:2"];  
string? versionOne = config["SupportedVersions:v1"];  
string? versionThree = config["SupportedVersions:v3"];  
  
// Write the values to the console.  
Console.WriteLine($"IPAddressRange:0 = {ipOne}");  
Console.WriteLine($"IPAddressRange:1 = {ipTwo}");  
Console.WriteLine($"IPAddressRange:2 = {ipThree}");  
Console.WriteLine($"SupportedVersions:v1 = {versionOne}");  
Console.WriteLine($"SupportedVersions:v3 = {versionThree}");
```

```
// Application code which might rely on the config could start here.

await host.RunAsync();

// This will output the following:
//     IPAddressRange:0 = 46.36.198.123
//     IPAddressRange:1 = 46.36.198.124
//     IPAddressRange:2 = 46.36.198.125
//     SupportedVersions:v1 = 1.0.0
//     SupportedVersions:v3 = 3.0.7
```

The values are accessed using the indexer API where each key is a string, and the value is a string. Configuration supports properties, objects, arrays, and dictionaries.

Configuration providers

The following table shows the configuration providers available to .NET Core apps.

| Provider | Provides configuration from |
|--|------------------------------------|
| Azure App configuration provider | Azure App Configuration |
| Azure Key Vault configuration provider | Azure Key Vault |
| Command-line configuration provider | Command-line parameters |
| Custom configuration provider | Custom source |
| Environment Variables configuration provider | Environment variables |
| File configuration provider | JSON, XML, and INI files |
| Key-per-file configuration provider | Directory files |
| Memory configuration provider | In-memory collections |
| App secrets (Secret Manager) | File in the user profile directory |

Tip

The order in which configuration providers are added matters. When multiple configuration providers are used and more than one provider specifies the same key, the last one added is used.

For more information on various configuration providers, see [Configuration providers in .NET](#).

See also

- Configuration providers in .NET
- [Implement a custom configuration provider](#)
- Configuration bugs should be created in the [github.com/dotnet/runtime ↗](https://github.com/dotnet/runtime) repo
- [Configuration in ASP.NET Core](#)

 [Collaborate with us on GitHub](#)

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

 .NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

Configuration providers in .NET

Article • 06/23/2023

Configuration in .NET is possible with configuration providers. Several types of providers rely on various configuration sources. This article details all of the different configuration providers and their corresponding sources.

- File configuration provider
- Environment variable configuration provider
- Command-line configuration provider
- Key-per-file configuration provider
- Memory configuration provider

File configuration provider

[FileConfigurationProvider](#) is the base class for loading configuration from the file system. The following configuration providers derive from `FileConfigurationProvider`:

- JSON configuration provider
- XML configuration provider
- INI configuration provider

Keys are case-insensitive. All of the file configuration providers throw the [FormatException](#) when duplicate keys are found in a single provider.

JSON configuration provider

The [JsonConfigurationProvider](#) class loads configuration from a JSON file. Install the [Microsoft.Extensions.Configuration.Json](#) NuGet package.

Overloads can specify:

- Whether the file is optional.
- Whether the configuration is reloaded if the file changes.

Consider the following code:

C#

```
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.Hosting;
using ConsoleJson.Example;
```

```

HostApplicationBuilder builder = Host.CreateApplicationBuilder(args);

builder.Configuration.Sources.Clear();

IHostEnvironment env = builder.Environment;

builder.Configuration
    .AddJsonFile("appsettings.json", optional: true, reloadOnChange: true)
    .AddJsonFile($"appsettings.{env.EnvironmentName}.json", true, true);

TransientFaultHandlingOptions options = new();
builder.Configuration.GetSection(nameof(TransientFaultHandlingOptions))
    .Bind(options);

Console.WriteLine($"TransientFaultHandlingOptions.Enabled= {options.Enabled}");
Console.WriteLine($"TransientFaultHandlingOptions.AutoRetryDelay= {options.AutoRetryDelay}");

using IHost host = builder.Build();

// Application code should start here.

await host.RunAsync();

```

The preceding code:

- Clears all existing configuration providers that were added by default in the `CreateApplicationBuilder(String[])` method.
- Configures the JSON configuration provider to load the `appsettings.json` and `appsettings.Environment.json` files with the following options:
 - `optional: true`: The file is optional.
 - `reloadOnChange: true`: The file is reloaded when changes are saved.

ⓘ Important

When [adding configuration providers](#) with `IConfigurationBuilder.Add`, the added configuration provider is added to the end of the end of the `IConfigurationSource` list. When keys are found by multiple providers, the last provider to read the key overrides previous providers.

An example `appsettings.json` file with various configuration settings follows:

JSON

```
{
  "SecretKey": "Secret key value",
```

```
"TransientFaultHandlingOptions": {  
    "Enabled": true,  
    "AutoRetryDelay": "00:00:07"  
},  
"Logging": {  
    "LogLevel": {  
        "Default": "Information",  
        "Microsoft": "Warning",  
        "Microsoft.Hosting.Lifetime": "Information"  
    }  
}  
}
```

From the [IConfigurationBuilder](#) instance, after configuration providers have been added, you can call [IConfigurationBuilder.Build\(\)](#) to get the [IConfigurationRoot](#) object. The configuration root represents the root of a configuration hierarchy. Sections from the configuration can be bound to instances of .NET objects and later provided as [IOptions<TOptions>](#) through dependency injection.

ⓘ Note

The *Build Action* and *Copy to Output Directory* properties of the JSON file must be set to *Content* and *Copy if newer (or Copy always)*, respectively.

Consider the `TransientFaultHandlingOptions` class defined as follows:

```
C#  
  
namespace ConsoleJson.Example;  
  
public sealed class TransientFaultHandlingOptions  
{  
    public bool Enabled { get; set; }  
    public TimeSpan AutoRetryDelay { get; set; }  
}
```

The following code builds the configuration root, binds a section to the `TransientFaultHandlingOptions` class type, and prints the bound values to the console window:

```
C#  
  
TransientFaultHandlingOptions options = new();  
builder.Configuration.GetSection(nameof(TransientFaultHandlingOptions))  
    .Bind(options);  
  
Console.WriteLine($"TransientFaultHandlingOptions.Enabled=
```

```
{options.Enabled}");  
Console.WriteLine($"TransientFaultHandlingOptions.AutoRetryDelay=  
{options.AutoRetryDelay}");
```

The application writes the following sample output:

C#

```
// Sample output:  
//     TransientFaultHandlingOptions.Enabled=True  
//     TransientFaultHandlingOptions.AutoRetryDelay=00:00:07
```

XML configuration provider

The [XmlConfigurationProvider](#) class loads configuration from an XML file at run time. Install the [Microsoft.Extensions.Configuration.Xml](#) NuGet package.

The following code demonstrates the configuration of XML files using the XML configuration provider.

C#

```
using Microsoft.Extensions.Configuration;  
using Microsoft.Extensions.Hosting;  
  
HostApplicationBuilder builder = Host.CreateApplicationBuilder(args);  
  
builder.Configuration.Sources.Clear();  
  
builder.Configuration  
    .AddXmlFile("appsettings.xml", optional: true, reloadOnChange: true)  
    .AddXmlFile("repeating-example.xml", optional: true, reloadOnChange:  
true);  
  
builder.Configuration.AddEnvironmentVariables();  
  
if (args is { Length: > 0 })  
{  
    builder.Configuration.AddCommandLine(args);  
}  
  
using IHost host = builder.Build();  
  
// Application code should start here.  
  
await host.RunAsync();
```

The preceding code:

- Clears all existing configuration providers that were added by default in the `CreateApplicationBuilder(String[])` method.
- Configures the XML configuration provider to load the `appsettings.xml` and `repeating-example.xml` files with the following options:
 - `optional: true`: The file is optional.
 - `reloadOnChange: true`: The file is reloaded when changes are saved.
- Configures the environment variables configuration provider.
- Configures the command-line configuration provider if the given `args` contain arguments.

The XML settings are overridden by settings in the [Environment variables configuration provider](#) and the [Command-line configuration provider](#).

An example `appsettings.xml` file with various configuration settings follows:

XML

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <SecretKey>Secret key value</SecretKey>
  <TransientFaultHandlingOptions>
    <Enabled>true</Enabled>
    <AutoRetryDelay>00:00:07</AutoRetryDelay>
  </TransientFaultHandlingOptions>
  <Logging>
    <LogLevel>
      <Default>Information</Default>
      <Microsoft>Warning</Microsoft>
    </LogLevel>
  </Logging>
</configuration>
```

Tip

To use the `IConfiguration` type in WinForms apps, add a reference to the [Microsoft.Extensions.Configuration.Xml](#) NuGet package. Instantiate the `ConfigurationBuilder` and chain calls to `AddXmlFile` and `Build()`. For more information, see [.NET Docs Issue #29679](#).

In .NET 5 and earlier versions, add the `name` attribute to distinguish repeating elements that use the same element name. In .NET 6 and later versions, the XML configuration provider automatically indexes repeating elements. That means you don't have to specify the `name` attribute, except if you want the "0" index in the key and there's only one element. (If you're upgrading to .NET 6 or later, you may encounter a break

resulting from this change in behavior. For more information, see [Repeated XML elements include index](#).)

XML

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
    <section name="section0">
        <key name="key0">value 00</key>
        <key name="key1">value 01</key>
    </section>
    <section name="section1">
        <key name="key0">value 10</key>
        <key name="key1">value 11</key>
    </section>
</configuration>
```

The following code reads the previous configuration file and displays the keys and values:

C#

```
IConfigurationRoot configurationRoot = builder.Configuration;

string key00 = "section:section0:key:key0";
string key01 = "section:section0:key:key1";
string key10 = "section:section1:key:key0";
string key11 = "section:section1:key:key1";

string? val00 = configurationRoot[key00];
string? val01 = configurationRoot[key01];
string? val10 = configurationRoot[key10];
string? val11 = configurationRoot[key11];

Console.WriteLine($"{key00} = {val00}");
Console.WriteLine($"{key01} = {val01}");
Console.WriteLine($"{key10} = {val10}");
Console.WriteLine($"{key11} = {val11}");
```

The application would write the following sample output:

C#

```
// Sample output:
//    section:section0:key:key0 = value 00
//    section:section0:key:key1 = value 01
//    section:section1:key:key0 = value 10
//    section:section1:key:key1 = value 11
```

Attributes can be used to supply values:

XML

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
  <key attribute="value" />
  <section>
    <key attribute="value" />
  </section>
</configuration>
```

The previous configuration file loads the following keys with `value`:

- `key:attribute`
- `section:key:attribute`

INI configuration provider

The `IniConfigurationProvider` class loads configuration from an INI file at run time. Install the [Microsoft.Extensions.Configuration.Ini](#) NuGet package.

The following code clears all the configuration providers and adds the `IniConfigurationProvider` with two INI files as the source:

C#

```
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.Hosting;

HostApplicationBuilder builder = Host.CreateApplicationBuilder(args);
builder.Configuration.Sources.Clear();

IHostEnvironment env = builder.Environment;

builder.Configuration
  .AddIniFile("appsettings.ini", optional: true, reloadOnChange: true)
  .AddIniFile($"appsettings.{env.EnvironmentName}.ini", true, true);

using IHost host = builder.Build();

// Application code should start here.

await host.RunAsync();
```

An example `appsettings.ini` file with various configuration settings follows:

ini

```
SecretKey="Secret key value"

[TransientFaultHandlingOptions]
Enabled=True
AutoRetryDelay="00:00:07"

[Logging:LogLevel]
Default=Information
Microsoft=Warning
```

The following code displays the preceding configuration settings by writing them to the console window:

C#

```
foreach ((string key, string? value) in
    builder.Configuration.AsEnumerable().Where(t => t.Value is not null))
{
    Console.WriteLine($"{key}={value}");
}
```

The application would write the following sample output:

C#

```
// Sample output:
// TransientFaultHandlingOptions:Enabled=True
// TransientFaultHandlingOptions:AutoRetryDelay=00:00:07
// SecretKey=Secret key value
// Logging:LogLevel:Microsoft=Warning
// Logging:LogLevel:Default=Information
```

Environment variable configuration provider

Using the default configuration, the [EnvironmentVariablesConfigurationProvider](#) loads configuration from environment variable key-value pairs after reading *appsettings.json*, *appsettings.Environment.json*, and Secret manager. Therefore, key values read from the environment override values read from *appsettings.json*, *appsettings.Environment.json*, and Secret manager.

The `:` delimiter doesn't work with environment variable hierarchical keys on all platforms. For example, the `:` delimiter is not supported by [Bash](#). The double underscore (`__`), which is supported on all platforms, automatically replaces any `:` delimiters in environment variables.

Consider the `TransientFaultHandlingOptions` class:

C#

```
public class TransientFaultHandlingOptions
{
    public bool Enabled { get; set; }
    public TimeSpan AutoRetryDelay { get; set; }
}
```

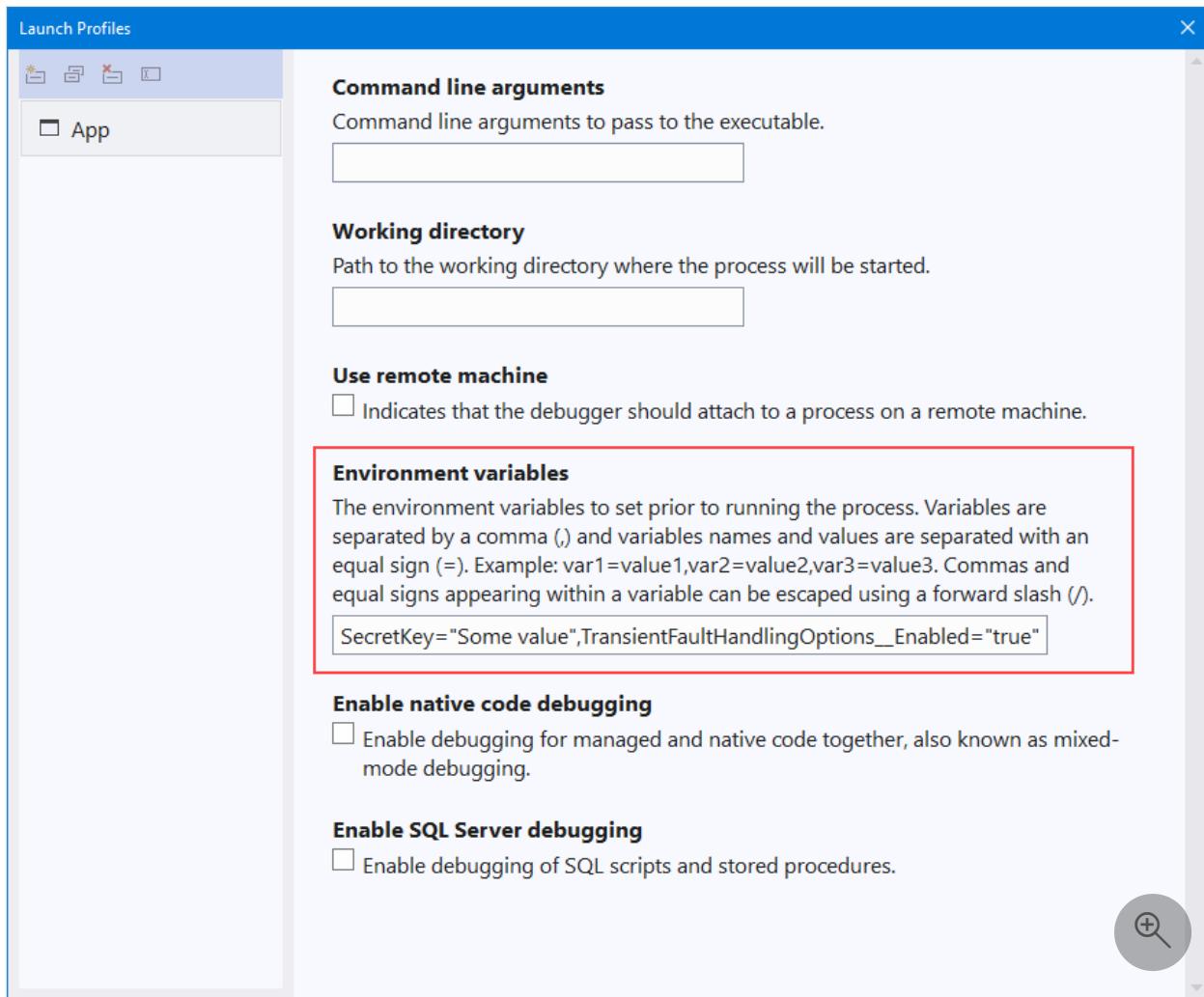
The following `set` commands set the environment keys and values of `SecretKey` and `TransientFaultHandlingOptions`.

.NET CLI

```
set SecretKey="Secret key from environment"
set TransientFaultHandlingOptions__Enabled="true"
set TransientFaultHandlingOptions__AutoRetryDelay="00:00:13"
```

These environment settings are only set in processes launched from the command window they were set in. They aren't read by web apps launched with Visual Studio.

With Visual Studio 2019 and later, you can specify environment variables using the **Launch Profiles** dialog.



The following `setx` commands can be used to set the environment keys and values on Windows. Unlike `set`, `setx` settings are persisted. `/M` sets the variable in the system environment. If the `/M` switch isn't used, a user environment variable is set.

.NET CLI

```
setx SecretKey "Secret key from setx environment" /M
setx TransientFaultHandlingOptions__Enabled "true" /M
setx TransientFaultHandlingOptions__AutoRetryDelay "00:00:05" /M
```

To test that the preceding commands override any `appsettings.json` and `appsettings.Environment.json` settings:

- With Visual Studio: Exit and restart Visual Studio.
- With the CLI: Start a new command window and enter `dotnet run`.

Prefixes

To specify a prefix for environment variables, call `AddEnvironmentVariables` with a string:

C#

```

using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.Hosting;

HostApplicationBuilder builder = Host.CreateApplicationBuilder(args);

builder.Configuration.AddEnvironmentVariables(prefix: "CustomPrefix_");
using IHost host = builder.Build();

// Application code should start here.

await host.RunAsync();

```

In the preceding code:

- `config.AddEnvironmentVariables(prefix: "CustomPrefix_")` is added after the default configuration providers. For an example of ordering the configuration providers, see [XML configuration provider](#).
- Environment variables set with the `CustomPrefix_` prefix override the default configuration providers. This includes environment variables without the prefix.

The prefix is stripped off when the configuration key-value pairs are read.

The default configuration loads environment variables and command-line arguments prefixed with `DOTNET_`. The `DOTNET_` prefix is used by .NET for [host](#) and [app configuration](#), but not for user configuration.

For more information on host and app configuration, see [.NET Generic Host](#).

Connection string prefixes

The Configuration API has special processing rules for four connection string environment variables. These connection strings are involved in configuring Azure connection strings for the app environment. Environment variables with the prefixes shown in the table are loaded into the app with the default configuration or when no prefix is supplied to `AddEnvironmentVariables`.

| Connection string prefix | Provider |
|-------------------------------|------------------------------------|
| <code>CUSTOMCONNSTR_</code> | Custom provider |
| <code>MYSQLCONNSTR_</code> | MySQL |
| <code>SQLAZURECONNSTR_</code> | Azure SQL Database |
| <code>SQLCONNSTR_</code> | SQL Server |

When an environment variable is discovered and loaded into configuration with any of the four prefixes shown in the table:

- The configuration key is created by removing the environment variable prefix and adding a configuration key section (`ConnectionStrings`).
- A new configuration key-value pair is created that represents the database connection provider (except for `CUSTOMCONNSTR_`, which has no stated provider).

| Environment variable key | Converted configuration key | Provider configuration entry |
|------------------------------------|--------------------------------------|---|
| <code>CUSTOMCONNSTR_{KEY}</code> | <code>ConnectionStrings:{KEY}</code> | Configuration entry not created. |
| <code>MYSQLCONNSTR_{KEY}</code> | <code>ConnectionStrings:{KEY}</code> | Key: <code>ConnectionStrings:{KEY}_ProviderName:</code> Value: <code>MySql.Data.MySqlClient</code> |
| <code>SQLAZURECONNSTR_{KEY}</code> | <code>ConnectionStrings:{KEY}</code> | Key: <code>ConnectionStrings:{KEY}_ProviderName:</code> Value: <code>System.Data.SqlClient</code> |
| <code>SQLCONNSTR_{KEY}</code> | <code>ConnectionStrings:{KEY}</code> | Key: <code>ConnectionStrings:{KEY}_ProviderName:</code> Value: <code>System.Data.SqlClient</code> |

Environment variables set in `launchSettings.json`

Environment variables set in `launchSettings.json` override those set in the system environment.

Azure App Service settings

On [Azure App Service](#), select **New application setting** on the **Settings > Configuration** page. Azure App Service application settings are:

- Encrypted at rest and transmitted over an encrypted channel.
- Exposed as environment variables.

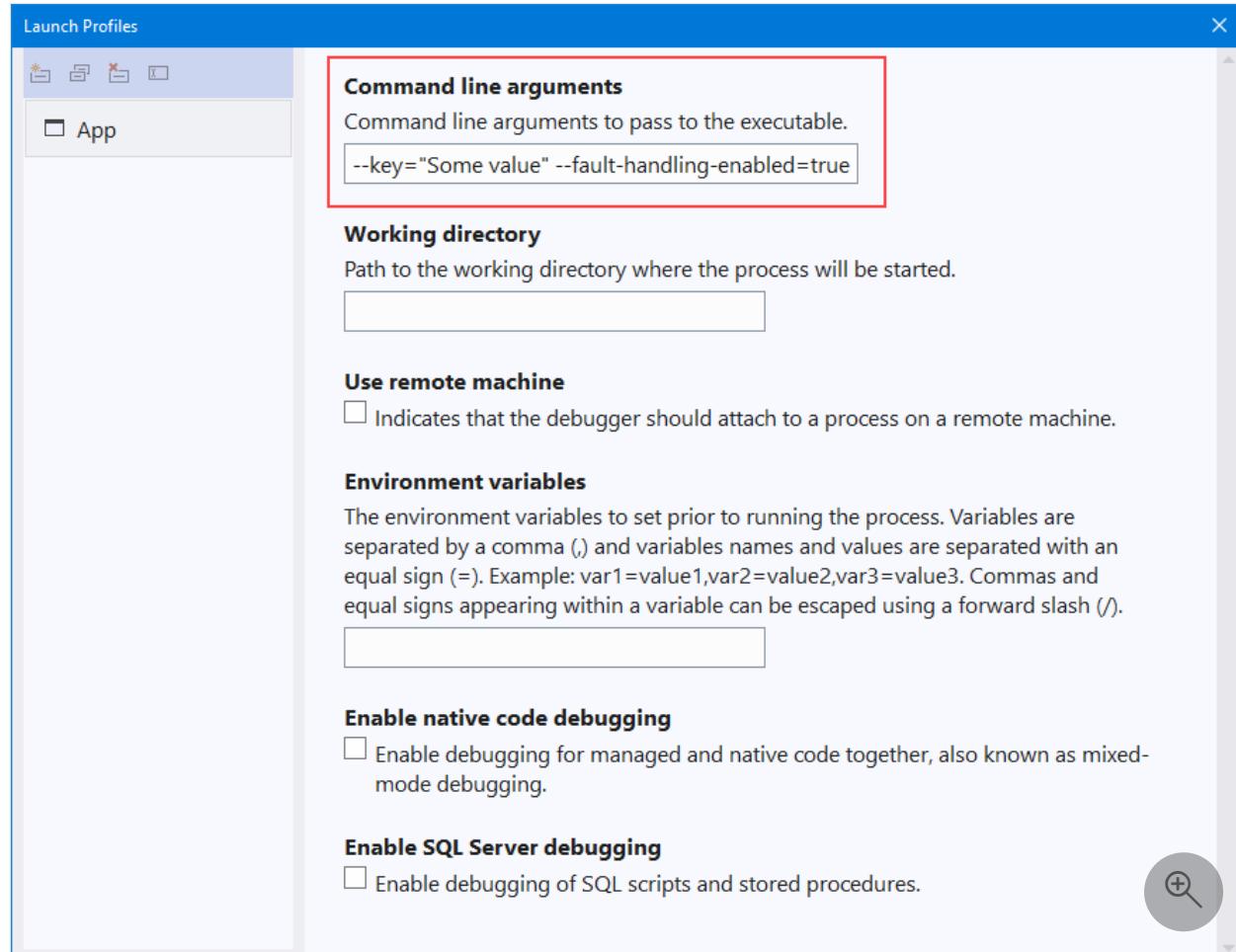
Command-line configuration provider

Using the default configuration, the [CommandLineConfigurationProvider](#) loads configuration from command-line argument key-value pairs after the following configuration sources:

- *appsettings.json* and *appsettings.Environment.json* files.
- App secrets (Secret Manager) in the `Development` environment.
- Environment variables.

By default, configuration values set on the command line override configuration values set with all the other configuration providers.

With Visual Studio 2019 and later, you can specify command-line arguments using the **Launch Profiles** dialog.



Command-line arguments

The following command sets keys and values using `=`:

```
.NET CLI
dotnet run SecretKey="Secret key from command line"
```

The following command sets keys and values using `/`:

```
.NET CLI
```

```
dotnet run /SecretKey "Secret key set from forward slash"
```

The following command sets keys and values using `--`:

.NET CLI

```
dotnet run --SecretKey "Secret key set from double hyphen"
```

The key value:

- Must follow `=`, or the key must have a prefix of `--` or `/` when the value follows a space.
- Isn't required if `=` is used. For example, `SomeKey=`.

Within the same command, don't mix command-line argument key-value pairs that use `=` with key-value pairs that use a space.

Key-per-file configuration provider

The [KeyPerFileConfigurationProvider](#) uses a directory's files as configuration key-value pairs. The key is the file name. The value is the file's contents. The Key-per-file configuration provider is used in Docker hosting scenarios.

To activate key-per-file configuration, call the [AddKeyPerFile](#) extension method on an instance of [ConfigurationBuilder](#). The `directoryPath` to the files must be an absolute path.

Overloads permit specifying:

- An `Action<KeyPerFileConfigurationSource>` delegate that configures the source.
- Whether the directory is optional and the path to the directory.

The double-underscore (`__`) is used as a configuration key delimiter in file names. For example, the file name `Logging__LogLevel__System` produces the configuration key `Logging:LogLevel:System`.

Call `ConfigureAppConfiguration` when building the host to specify the app's configuration:

C#

```
.ConfigureAppConfiguration((_, configuration) =>
{
```

```
    var path = Path.Combine(
        Directory.GetCurrentDirectory(), "path/to/files");

    configuration.AddKeyPerFile(directoryPath: path, optional: true);
}
```

Memory configuration provider

The [MemoryConfigurationProvider](#) uses an in-memory collection as configuration key-value pairs.

The following code adds a memory collection to the configuration system:

C#

```
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.Hosting;

HostApplicationBuilder builder = Host.CreateApplicationBuilder(args);

builder.Configuration.AddInMemoryCollection(
    new Dictionary<string, string?>
{
    ["SecretKey"] = "Dictionary MyKey Value",
    ["TransientFaultHandlingOptions:Enabled"] = bool.TrueString,
    ["TransientFaultHandlingOptions:AutoRetryDelay"] = "00:00:07",
    ["Logging:LogLevel:Default"] = "Warning"
});

using IHost host = builder.Build();

// Application code should start here.

await host.RunAsync();
```

In the preceding code,

[MemoryConfigurationBuilderExtensions.AddInMemoryCollection\(IConfigurationBuilder, IEnumerable<KeyValuePair<String, String>>\)](#) adds the memory provider after the default configuration providers. For an example of ordering the configuration providers, see [XML configuration provider](#).

See also

- [Configuration in .NET](#)
- [.NET Generic Host](#)
- [Implement a custom configuration provider](#)

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

Implement a custom configuration provider in .NET

Article • 06/23/2023

There are many [configuration providers](#) available for common configuration sources such as JSON, XML, and INI files. You may need to implement a custom configuration provider when one of the available providers doesn't suit your application needs. In this article, you'll learn how to implement a custom configuration provider that relies on a database as its configuration source.

Custom configuration provider

The sample app demonstrates how to create a basic configuration provider that reads configuration key-value pairs from a database using [Entity Framework \(EF\) Core](#).

The provider has the following characteristics:

- The EF in-memory database is used for demonstration purposes.
 - To use a database that requires a connection string, get a connection string from an interim configuration.
- The provider reads a database table into configuration at startup. The provider doesn't query the database on a per-key basis.
- Reload-on-change isn't implemented, so updating the database after the app has started will not affect the app's configuration.

Define a `Settings` record type entity for storing configuration values in the database.

For example, you could add a `Settings.cs` file in your `Models` folder:

```
C#  
  
namespace CustomProvider.Example.Models;  
  
public record Settings(string Id, string? Value);
```

For information on record types, see [Record types in C#](#).

Add an `EntityConfigurationContext` to store and access the configured values.

Providers/EntityConfigurationContext.cs:

```
C#
```

```

using CustomProvider.Example.Models;
using Microsoft.EntityFrameworkCore;

namespace CustomProvider.Example.Providers;

public class EntityConfigurationContext : DbContext
{
    private readonly string _connectionString;

    public DbSet<Settings> Settings => Set<Settings>();

    public EntityConfigurationContext(string? connectionString) =>
        _connectionString = connectionString ?? "";

    protected override void OnConfiguring(DbContextOptionsBuilder
optionsBuilder)
    {
        _ = _connectionString switch
        {
            { Length: > 0 } =>
optionsBuilder.UseSqlServer(_connectionString),
            _ => optionsBuilder.UseInMemoryDatabase("InMemoryDatabase")
        };
    }
}

```

By overriding [OnConfiguring\(DbContextOptionsBuilder\)](#) you can use the appropriate database connection. For example, if a connection string was provided you could connect to SQL Server, otherwise you could rely on an in-memory database.

Create a class that implements [IConfigurationSource](#).

Providers/EntityConfigurationSource.cs:

C#

```

using Microsoft.Extensions.Configuration;

namespace CustomProvider.Example.Providers;

public class EntityConfigurationSource : IConfigurationSource
{
    private readonly string? _connectionString;

    public EntityConfigurationSource(string? connectionString) =>
        _connectionString = connectionString;

    public IConfigurationProvider Build(IConfigurationBuilder builder) =>
        new EntityConfigurationProvider(_connectionString);
}

```

Create the custom configuration provider by inheriting from [ConfigurationProvider](#). The configuration provider initializes the database when it's empty. Since configuration keys are case-insensitive, the dictionary used to initialize the database is created with the case-insensitive comparer ([StringComparer.OrdinalIgnoreCase](#)).

Providers/EntityConfigurationProvider.cs:

C#

```
using CustomProvider.Example.Models;
using Microsoft.Extensions.Configuration;

namespace CustomProvider.Example.Providers;

public class EntityConfigurationProvider : ConfigurationProvider
{
    private readonly string? _connectionString;

    public EntityConfigurationProvider(string? connectionString) =>
        _connectionString = connectionString;

    public override void Load()
    {
        using var dbContext = new
EntityConfigurationContext(_connectionString);

        dbContext.Database.EnsureCreated();

        Data = dbContext.Settings.Any()
            ? dbContext.Settings.ToDictionary<Settings, string, string?>(c
=> c.Id, c => c.Value, StringComparer.OrdinalIgnoreCase)
            : CreateAndSaveDefaultValues(dbContext);
    }

    static IDictionary<string, string?> CreateAndSaveDefaultValues(
        EntityConfigurationContext context)
    {
        var settings = new Dictionary<string, string?>(
            StringComparer.OrdinalIgnoreCase)
        {
            ["WidgetOptions:EndpointId"] = "b3da3c4c-9c4e-4411-bc4d-
609e2dcc5c67",
            ["WidgetOptions:DisplayLabel"] = "Widgets Incorporated, LLC.",
            ["WidgetOptions:WidgetRoute"] = "api/widgets"
        };

        context.Settings.AddRange(
            settings.Select(kvp => new Settings(kvp.Key, kvp.Value))
                .ToArray());
    }

    context.SaveChanges();

    return settings;
}
```

```
    }  
}
```

An `AddEntityConfiguration` extension method permits adding the configuration source to a `IConfigurationBuilder` instance.

Extensions/ConfigurationBuilderExtensions.cs:

C#

```
using CustomProvider.Example.Providers;  
  
namespace Microsoft.Extensions.Configuration;  
  
public static class ConfigurationBuilderExtensions  
{  
    public static IApplicationBuilder AddEntityConfiguration(  
        this IApplicationBuilder builder)  
    {  
        var tempConfig = builder.Build();  
        var connectionString =  
            tempConfig.GetConnectionString("WidgetConnectionString");  
  
        return builder.Add(new EntityConfigurationSource(connectionString));  
    }  
}
```

ⓘ Important

The use of a temporary configuration source to acquire the connection string is important. The current `builder` has its configuration constructed temporarily by calling `IApplicationBuilder.Build()`, and `GetConnectionString`. After obtaining the connection string, the `builder` adds the `EntityConfigurationSource` given the `connectionString`.

The following code shows how to use the custom `EntityConfigurationProvider` in `Program.cs`:

C#

```
using Microsoft.Extensions.Configuration;  
using Microsoft.Extensions.DependencyInjection;  
using Microsoft.Extensions.Hosting;  
using Microsoft.Extensions.Options;  
using CustomProvider.Example;  
  
HostApplicationBuilder builder = Host.CreateApplicationBuilder(args);
```

```
builder.Configuration.Sources.Clear();
builder.Configuration.AddEntityConfiguration();
builder.Services.Configure<WidgetOptions>(
    builder.Configuration.GetSection("WidgetOptions"));

using IHost host = builder.Build();

WidgetOptions options =
host.Services.GetRequiredService<IOptions<WidgetOptions>>().Value;
Console.WriteLine($"DisplayLabel={options.DisplayLabel}");
Console.WriteLine($"EndpointId={options.EndpointId}");
Console.WriteLine($"WidgetRoute={options.WidgetRoute}");

await host.RunAsync();
// Sample output:
//   WidgetRoute=api/widgets
//   EndpointId=b3da3c4c-9c4e-4411-bc4d-609e2dcc5c67
//   DisplayLabel=Widgets Incorporated, LLC.
```

Consume provider

To consume the custom configuration provider, you can use the [options pattern](#). With the sample app in place, define an options object to represent the widget settings.

C#

```
namespace CustomProvider.Example;

public class WidgetOptions
{
    public required Guid EndpointId { get; set; }

    public required string DisplayLabel { get; set; } = null!;

    public required string WidgetRoute { get; set; } = null!;
}
```

A call to [Configure](#) registers a configuration instance, which `IOptions<WidgetOptions>` binds against.

C#

```
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
using Microsoft.Extensions.Options;
using CustomProvider.Example;

HostApplicationBuilder builder = Host.CreateApplicationBuilder(args);
```

```

builder.Configuration.Sources.Clear();
builder.Configuration.AddEntityConfiguration();
builder.Services.Configure<WidgetOptions>(
    builder.Configuration.GetSection("WidgetOptions"));

using IHost host = builder.Build();

WidgetOptions options =
host.Services.GetRequiredService<IOptions<WidgetOptions>>().Value;
Console.WriteLine($"DisplayLabel={options.DisplayLabel}");
Console.WriteLine($"EndpointId={options.EndpointId}");
Console.WriteLine($"WidgetRoute={options.WidgetRoute}");

await host.RunAsync();
// Sample output:
//   WidgetRoute=api/widgets
//   EndpointId=b3da3c4c-9c4e-4411-bc4d-609e2dcc5c67
//   DisplayLabel=Widgets Incorporated, LLC.

```

The preceding code configures the `WidgetOptions` object from the "WidgetOptions" section of the configuration. This enables the options pattern, exposing a dependency injection-ready `IOptions<WidgetOptions>` representation of the EF settings. The options are ultimately provided from the custom configuration provider.

See also

- [Configuration in .NET](#)
- [Configuration providers in .NET](#)
- [Options pattern in .NET](#)
- [Dependency injection in .NET](#)

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

Options pattern in .NET

Article • 06/23/2023

The options pattern uses classes to provide strongly-typed access to groups of related settings. When [configuration settings](#) are isolated by scenario into separate classes, the app adheres to two important software engineering principles:

- The [Interface Segregation Principle \(ISP\)](#) or [Encapsulation](#): Scenarios (classes) that depend on configuration settings depend only on the configuration settings that they use.
- [Separation of Concerns](#): Settings for different parts of the app aren't dependent or coupled with one another.

Options also provide a mechanism to validate configuration data. For more information, see the [Options validation](#) section.

Bind hierarchical configuration

The preferred way to read related configuration values is using the options pattern. The options pattern is possible through the `IOptions<TOptions>` interface, where the generic type parameter `TOptions` is constrained to a `class`. The `IOptions<TOptions>` can later be provided through dependency injection. For more information, see [Dependency injection in .NET](#).

For example, to read the highlighted configuration values from an `appsettings.json` file:

```
JSON

{
    "SecretKey": "Secret key value",
    "TransientFaultHandlingOptions": {
        "Enabled": true,
        "AutoRetryDelay": "00:00:07"
    },
    "Logging": {
        "LogLevel": {
            "Default": "Information",
            "Microsoft": "Warning",
            "Microsoft.Hosting.Lifetime": "Information"
        }
    }
}
```

Create the following `TransientFaultHandlingOptions` class:

C#

```
public sealed class TransientFaultHandlingOptions
{
    public bool Enabled { get; set; }
    public TimeSpan AutoRetryDelay { get; set; }
}
```

When using the options pattern, an options class:

- Must be non-abstract with a public parameterless constructor
- Contain public read-write properties to bind (fields are **not** bound)

The following code is part of the *Program.cs* C# file and:

- Calls [ConfigurationBinder.Bind](#) to bind the `TransientFaultHandlingOptions` class to the `"TransientFaultHandlingOptions"` section.
- Displays the configuration data.

C#

```
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.Hosting;
using ConsoleJson.Example;

HostApplicationBuilder builder = Host.CreateApplicationBuilder(args);

builder.Configuration.Sources.Clear();

IHostEnvironment env = builder.Environment;

builder.Configuration
    .AddJsonFile("appsettings.json", optional: true, reloadOnChange: true)
    .AddJsonFile($"appsettings.{env.EnvironmentName}.json", true, true);

TransientFaultHandlingOptions options = new();
builder.Configuration.GetSection(nameof(TransientFaultHandlingOptions))
    .Bind(options);

Console.WriteLine($"TransientFaultHandlingOptions.Enabled={options.Enabled}");
Console.WriteLine($"TransientFaultHandlingOptions.AutoRetryDelay={options.AutoRetryDelay}");

using IHost host = builder.Build();

// Application code should start here.

await host.RunAsync();
```

```
// <Output>
// Sample output:
```

In the preceding code, the JSON configuration file has its `"TransientFaultHandlingOptions"` section bound to the `TransientFaultHandlingOptions` instance. This hydrates the C# objects properties with those corresponding values from the configuration.

`ConfigurationBinder.Get<T>` binds and returns the specified type. `ConfigurationBinder.Get<T>` may be more convenient than using `ConfigurationBinder.Bind`. The following code shows how to use `ConfigurationBinder.Get<T>` with the `TransientFaultHandlingOptions` class:

C#

```
var options =
    builder.Configuration.GetSection(nameof(TransientFaultHandlingOptions))
    .Get<TransientFaultHandlingOptions>();

Console.WriteLine($"TransientFaultHandlingOptions.Enabled=
{options.Enabled}");
Console.WriteLine($"TransientFaultHandlingOptions.AutoRetryDelay=
{options.AutoRetryDelay}");
```

In the preceding code, the `ConfigurationBinder.Get<T>` is used to acquire an instance of the `TransientFaultHandlingOptions` object with its property values populated from the underlying configuration.

ⓘ Important

The `ConfigurationBinder` class exposes several APIs, such as `.Bind(object instance)` and `.Get<T>()` that are *not* constrained to `class`. When using any of the **Options interfaces**, you must adhere to aforementioned **options class constraints**.

An alternative approach when using the options pattern is to bind the `"TransientFaultHandlingOptions"` section and add it to the [dependency injection service container](#). In the following code, `TransientFaultHandlingOptions` is added to the service container with `Configure` and bound to configuration:

C#

```
HostApplicationBuilder builder = Host.CreateApplicationBuilder(args);
```

```
builder.Services.Configure<TransientFaultHandlingOptions>(
    builder.Configuration.GetSection(
        key: nameof(TransientFaultHandlingOptions)));
```

The `builder` in the preceding example is an instance of [HostApplicationBuilder](#).

💡 Tip

The `key` parameter is the name of the configuration section to search for. It does *not* have to match the name of the type that represents it. For example, you could have a section named `"FaultHandling"` and it could be represented by the `TransientFaultHandlingOptions` class. In this instance, you'd pass `"FaultHandling"` to the `GetSection` function instead. The `nameof` operator is used as a convenience when the named section matches the type it corresponds to.

Using the preceding code, the following code reads the position options:

C#

```
using Microsoft.Extensions.Options;

namespace ConsoleJson.Example;

public sealed class ExampleService
{
    private readonly TransientFaultHandlingOptions _options;

    public ExampleService(IOptions<TransientFaultHandlingOptions> options)
=>
    _options = options.Value;

    public void DisplayValues()
    {
        Console.WriteLine($"TransientFaultHandlingOptions.Enabled={_
            _options.Enabled}");
        Console.WriteLine($"TransientFaultHandlingOptions.AutoRetryDelay=_
            {_options.AutoRetryDelay}");
    }
}
```

In the preceding code, changes to the JSON configuration file after the app has started are *not* read. To read changes after the app has started, use [IOptionsSnapshot](#) or [IOptionsMonitor](#) to monitor changes as they occur, and react accordingly.

Options interfaces

[IOptions<TOptions>:](#)

- Does **not** support:
 - Reading of configuration data after the app has started.
 - [Named options](#)
- Is registered as a [Singleton](#) and can be injected into any [service lifetime](#).

[IOptionsSnapshot<TOptions>:](#)

- Is useful in scenarios where options should be recomputed on every injection resolution, in [scoped or transient lifetimes](#). For more information, see [Use IOptionsSnapshot to read updated data](#).
- Is registered as [Scoped](#) and therefore cannot be injected into a Singleton service.
- Supports [named options](#)

[IOptionsMonitor<TOptions>:](#)

- Is used to retrieve options and manage options notifications for `TOptions` instances.
- Is registered as a [Singleton](#) and can be injected into any [service lifetime](#).
- Supports:
 - Change notifications
 - [Named options](#)
 - [Reloadable configuration](#)
 - Selective options invalidation ([IOptionsMonitorCache<TOptions>](#))

[IOptionsFactory<TOptions>](#) is responsible for creating new options instances. It has a single [Create](#) method. The default implementation takes all registered [IConfigureOptions<TOptions>](#) and [IPostConfigureOptions<TOptions>](#) and runs all the configurations first, followed by the post-configuration. It distinguishes between [IConfigureNamedOptions<TOptions>](#) and [IConfigureOptions<TOptions>](#) and only calls the appropriate interface.

[IOptionsMonitorCache<TOptions>](#) is used by [IOptionsMonitor<TOptions>](#) to cache `TOptions` instances. The [IOptionsMonitorCache<TOptions>](#) invalidates options instances in the monitor so that the value is recomputed ([TryRemove](#)). Values can be manually introduced with [TryAdd](#). The [Clear](#) method is used when all named instances should be recreated on demand.

[IOptionsChangeTokenSource<TOptions>](#) is used to fetch the [IChangeToken](#) that tracks changes to the underlying `TOptions` instance. For more information on change-token primitives, see [Change notifications](#).

Options interfaces benefits

Using a generic wrapper type gives you the ability to decouple the lifetime of the option from the DI container. The `IOptions<TOptions>.Value` interface provides a layer of abstraction, including generic constraints, on your options type. This provides the following benefits:

- The evaluation of the `T` configuration instance is deferred to the accessing of `IOptions<TOptions>.Value`, rather than when it is injected. This is important because you can consume the `T` option from various places and choose the lifetime semantics without changing anything about `T`.
- When registering options of type `T`, you do not need to explicitly register the `T` type. This is a convenience when you're [authoring a library](#) with simple defaults, and you don't want to force the caller to register options into the DI container with a specific lifetime.
- From the perspective of the API, it allows for constraints on the type `T` (in this case, `T` is constrained to a reference type).

Use `IOptionsSnapshot` to read updated data

When you use `IOptionsSnapshot<TOptions>`, options are computed once per request when accessed and are cached for the lifetime of the request. Changes to the configuration are read after the app starts when using configuration providers that support reading updated configuration values.

The difference between `IOptionsMonitor` and `IOptionsSnapshot` is that:

- `IOptionsMonitor` is a [singleton service](#) that retrieves current option values at any time, which is especially useful in singleton dependencies.
- `IOptionsSnapshot` is a [scoped service](#) and provides a snapshot of the options at the time the `IOptionsSnapshot<T>` object is constructed. Options snapshots are designed for use with transient and scoped dependencies.

The following code uses `IOptionsSnapshot<TOptions>`.

C#

```
using Microsoft.Extensions.Options;

namespace ConsoleJson.Example;

public sealed class ScopedService
{
```

```
private readonly TransientFaultHandlingOptions _options;

public ScopedService(IOptionsSnapshot<TransientFaultHandlingOptions>
options) =>
    _options = options.Value;

public void DisplayValues()
{
    Console.WriteLine($"TransientFaultHandlingOptions.Enabled={_
options.Enabled}");
    Console.WriteLine($"TransientFaultHandlingOptions.AutoRetryDelay=_
options.AutoRetryDelay");
}

}
```

The following code registers a configuration instance which `TransientFaultHandlingOptions` binds against:

C#

```
builder.Services
    .Configure<TransientFaultHandlingOptions>(
        configurationRoot.GetSection(
            nameof(TransientFaultHandlingOptions)));
```

In the preceding code, the `Configure<TOptions>` method is used to register a configuration instance that `TOptions` will bind against, and updates the options when the configuration changes.

IOptionsMonitor

To use the options monitor, options objects are configured in the same way from a configuration section.

C#

```
builder.Services
    .Configure<TransientFaultHandlingOptions>(
        configurationRoot.GetSection(
            nameof(TransientFaultHandlingOptions)));
```

The following example uses `IOptionsMonitor<TOptions>`:

C#

```
using Microsoft.Extensions.Options;
```

```

namespace ConsoleJson.Example;

public sealed class MonitorService
{
    private readonly IOptionsMonitor<TransientFaultHandlingOptions>
    _monitor;

    public MonitorService(IOptionsMonitor<TransientFaultHandlingOptions>
    monitor) =>
        _monitor = monitor;

    public void DisplayValues()
    {
        TransientFaultHandlingOptions options = _monitor.CurrentValue;

        Console.WriteLine($"TransientFaultHandlingOptions.Enabled=
{options.Enabled}");
        Console.WriteLine($"TransientFaultHandlingOptions.AutoRetryDelay=
{options.AutoRetryDelay}");
    }
}

```

In the preceding code, changes to the JSON configuration file after the app has started are read.

Tip

Some file systems, such as Docker containers and network shares, may not reliably send change notifications. When using the `IOptionsMonitor<TOptions>` interface in these environments, set the `DOTNET_USE_POLLING_FILE_WATCHER` environment variable to `1` or `true` to poll the file system for changes. The interval at which changes are polled is every four seconds and is not configurable.

For more information on Docker containers, see [Containerize a .NET app](#).

Named options support using `IConfigureNamedOptions`

Named options:

- Are useful when multiple configuration sections bind to the same properties.
- Are case-sensitive.

Consider the following `appsettings.json` file:

JSON

```
{  
    "Features": {  
        "Personalize": {  
            "Enabled": true,  
            "ApiKey":  
                "aGEgaGEgeW91IHRob3VnaHQgdGhhCB3YXMgcwVhbGx5IHNvbW0aGluZw=="  
        },  
        "WeatherStation": {  
            "Enabled": true,  
            "ApiKey": "QXJlIH1vdSBhdHR1bXB0aW5nIHRvIGhhY2sgdXM/"  
        }  
    }  
}
```

Rather than creating two classes to bind `Features:Personalize` and `Features:WeatherStation`, the following class is used for each section:

C#

```
public class Features  
{  
    public const string Personalize = nameof(Personalize);  
    public const string WeatherStation = nameof(WeatherStation);  
  
    public bool Enabled { get; set; }  
    public string ApiKey { get; set; }  
}
```

The following code configures the named options:

C#

```
HostApplicationBuilder builder = Host.CreateApplicationBuilder(args);  
  
// Omitted for brevity...  
  
builder.Services.Configure<Features>(  
    Features.Personalize,  
    builder.Configuration.GetSection("Features:Personalize"));  
  
builder.Services.Configure<Features>(  
    Features.WeatherStation,  
    builder.Configuration.GetSection("Features:WeatherStation"));
```

The following code displays the named options:

C#

```

public class sealed Service
{
    private readonly Features _personalizeFeature;
    private readonly Features _weatherStationFeature;

    public Service(IOptionsSnapshot<Features> namedOptionsAccessor)
    {
        _personalizeFeature =
            namedOptionsAccessor.Get(Features.Personalize);
        _weatherStationFeature =
            namedOptionsAccessor.Get(Features.WeatherStation);
    }
}

```

All options are named instances. `IConfigureOptions<TOptions>` instances are treated as targeting the `Options.DefaultName` instance, which is `string.Empty`.

`IConfigureNamedOptions<TOptions>` also implements `IConfigureOptions<TOptions>`. The default implementation of the `IOptionsFactory<TOptions>` has logic to use each appropriately. The `null` named option is used to target all of the named instances instead of a specific named instance. `ConfigureAll` and `PostConfigureAll` use this convention.

OptionsBuilder API

`OptionsBuilder<TOptions>` is used to configure `TOptions` instances. `OptionsBuilder` streamlines creating named options as it's only a single parameter to the initial `AddOptions<TOptions>(string optionsName)` call instead of appearing in all of the subsequent calls. Options validation and the `ConfigureOptions` overloads that accept service dependencies are only available via `OptionsBuilder`.

`OptionsBuilder` is used in the [Options validation](#) section.

Use DI services to configure options

Services can be accessed from dependency injection while configuring options in two ways:

- Pass a configuration delegate to `Configure` on `OptionsBuilder<TOptions>`. `OptionsBuilder<TOptions>` provides overloads of `Configure` that allow use of up to five services to configure options:

C#

```
builder.Services
    .AddOptions<MyOptions>("optionalName")
    .Configure<ExampleService, ScopedService, MonitorService>(
        (options, es, ss, ms) =>
            options.Property = DoSomethingWith(es, ss, ms));
```

- Create a type that implements [IConfigureOptions<TOptions>](#) or [IConfigureNamedOptions<TOptions>](#) and register the type as a service.

We recommend passing a configuration delegate to [Configure](#), since creating a service is more complex. Creating a type is equivalent to what the framework does when calling [Configure](#). Calling [Configure](#) registers a transient generic [IConfigureNamedOptions<TOptions>](#), which has a constructor that accepts the generic service types specified.

Options validation

Options validation enables option values to be validated.

Consider the following *appsettings.json* file:

JSON

```
{  
    "MyCustomSettingsSection": {  
        "SiteTitle": "Amazing docs from Awesome people!",  
        "Scale": 10,  
        "VerbosityLevel": 32  
    }  
}
```

The following class binds to the `"MyCustomSettingsSection"` configuration section and applies a couple of `DataAnnotations` rules:

C#

```
using System.ComponentModel.DataAnnotations;  
  
namespace ConsoleJson.Example;  
  
public sealed class SettingsOptions  
{  
    public const string ConfigurationSectionName =  
        "MyCustomSettingsSection";  
  
    [RegularExpression(@"^[\w\W]{1,40}$")]
```

```
    public required string SiteTitle { get; set; } = null!;

    [Range(0, 1000,
        ErrorMessage = "Value for {0} must be between {1} and {2}.")]
    public required int Scale { get; set; }

    public required int VerbosityLevel { get; set; }
}
```

In the preceding `SettingsOptions` class, the `ConfigurationSectionName` property contains the name of the configuration section to bind to. In this scenario, the options object provides the name of its configuration section.

💡 Tip

The configuration section name is independent of the configuration object that it's binding to. In other words, a configuration section named `"FooBarOptions"` can be bound to an options object named `ZedOptions`. Although it might be common to name them the same, it's *not* necessary and can actually cause name conflicts.

The following code:

- Calls `AddOptions` to get an `OptionsResolver<TOptions>` that binds to the `SettingsOptions` class.
- Calls `ValidateDataAnnotations` to enable validation using `DataAnnotations`.

C#

```
builder.Services
    .AddOptions<SettingsOptions>()

    .Bind(Configuration.GetSection(SettingsOptions.ConfigurationSectionName))
    .ValidateDataAnnotations();
```

The `ValidateDataAnnotations` extension method is defined in the [Microsoft.Extensions.Options.DataAnnotations](#) NuGet package.

The following code displays the configuration values or the validation errors:

C#

```
using Microsoft.Extensions.Logging;
using Microsoft.Extensions.Options;

namespace ConsoleJson.Example;
```

```

public sealed class ValidationService
{
    private readonly ILogger<ValidationService> _logger;
    private readonly IOptions<SettingsOptions> _config;

    public ValidationService(
        ILogger<ValidationService> logger,
        IOptions<SettingsOptions> config)
    {
        _config = config;
        _logger = logger;

        try
        {
            SettingsOptions options = _config.Value;
        }
        catch (OptionsValidationException ex)
        {
            foreach (string failure in ex.Failures)
            {
                _logger.LogError(failure);
            }
        }
    }
}

```

The following code applies a more complex validation rule using a delegate:

```

C#

builder.Services
    .AddOptions<SettingsOptions>()

    .Bind(Configuration.GetSection(SettingsOptions.ConfigurationSectionName))
        .ValidateDataAnnotations()
        .Validate(config =>
    {
        if (config.Scale != 0)
        {
            return config.VerboseLevel > config.Scale;
        }

        return true;
    }, "VerboseLevel must be > than Scale.");

```

IValidateOptions for complex validation

The following class implements [IValidateOptions<TOptions>](#):

C#

```
using System.Text;
using System.Text.RegularExpressions;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.Options;

namespace ConsoleJson.Example;

sealed partial class ValidateSettingsOptions : 
IValidateOptions<SettingsOptions>
{
    public SettingsOptions? _settings { get; private set; }

    public ValidateSettingsOptions(IConfiguration config) =>
        _settings =
config.GetSection(SettingsOptions.ConfigurationSectionName)
            .Get<SettingsOptions>();

    public ValidateOptionsResult Validate(string? name, SettingsOptions options)
    {
        StringBuilder? failure = null;

        if (!ValidationRegex().IsMatch(options.SiteTitle))
        {
            (failure ??= new()).AppendLine($"{options.SiteTitle} doesn't
match RegEx");
        }

        if (options.Scale is < 0 or > 1_000)
        {
            (failure ??= new()).AppendLine($"{options.Scale} isn't within
Range 0 - 1000");
        }

        if (_settings is { Scale: 0 } && _settings.VerboseLevel <=
_settings.Scale)
        {
            (failure ??= new()).AppendLine("VerboseLevel must be > than
Scale.");
        }

        return failure is not null ?
            ? ValidateOptionsResult.Fail(failure.ToString())
            : ValidateOptionsResult.Success;
    }

    [GeneratedRegex("^[a-zA-Z'-'\\s]{1,40}$")]
    private static partial Regex ValidationRegex();
}
```

IValidateOptions enables moving the validation code into a class.

ⓘ Note

This example code relies on the [Microsoft.Extensions.Configuration.Json](#) NuGet package.

Using the preceding code, validation is enabled when configuring services with the following code:

C#

```
HostApplicationBuilder builder = Host.CreateApplicationBuilder(args);

// Omitted for brevity...

builder.Services.Configure<SettingsOptions>(
    builder.Configuration.GetSection(
        SettingsOptions.ConfigurationSectionName));

builder.Services.TryAddEnumerable(
    ServiceDescriptor.Singleton
    <IValidateOptions<SettingsOptions>, ValidateSettingsOptions>());
```

Options post-configuration

Set post-configuration with [IPostConfigureOptions<TOptions>](#). Post-configuration runs after all [IConfigureOptions<TOptions>](#) configuration occurs, and can be useful in scenarios when you need to override configuration:

C#

```
builder.Services.PostConfigure<CustomOptions>(customOptions =>
{
    customOptions.Option1 = "post_configured_option1_value";
});
```

[PostConfigure](#) is available to post-configure named options:

C#

```
builder.Services.PostConfigure<CustomOptions>("named_options_1",
customOptions =>
{
    customOptions.Option1 = "post_configured_option1_value";
});
```

Use `PostConfigureAll` to post-configure all configuration instances:

C#

```
builder.Services.PostConfigureAll<CustomOptions>(customOptions =>
{
    customOptions.Option1 = "post_configured_option1_value";
});
```

See also

- [Configuration in .NET](#)
- [Options pattern guidance for .NET library authors](#)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

Compile-time options validation source generation

Article • 12/20/2023

In the [options pattern](#), various methods for validating options are presented. These methods include using data annotation attributes or employing a custom validator. Data annotation attributes are validated at run time and can incur performance costs. This article demonstrates how to utilize the options validation source generator to produce optimized validation code at compile time.

Automatic `IValidateOptions` implementation generation

The [options pattern](#) article illustrates how to implement the `IValidateOptions<TOptions>` interface for validating options. The options validation source generator can automatically create the `IValidateOptions` interface implementation by leveraging data annotation attributes on the options class.

The content that follows takes the annotation attributes example that's shown in [Options pattern](#) and converts it to use the options validation source generator.

Consider the following `appsettings.json` file:

```
JSON

{
  "MyCustomSettingsSection": {
    "SiteTitle": "Amazing docs from awesome people!",
    "Scale": 10,
    "VerbosityLevel": 32
  }
}
```

The following class binds to the `"MyCustomSettingsSection"` configuration section and applies a couple of `DataAnnotations` rules:

```
C#

using System.ComponentModel.DataAnnotations;

namespace ConsoleJson.Example;
```

```

public sealed class SettingsOptions
{
    public const string ConfigurationSectionName =
"MyCustomSettingsSection";

    [Required]
    [RegularExpression(@"^a-zA-Z'-'s]{1,40}$")]
    public required string SiteTitle { get; set; }

    [Required]
    [Range(0, 1_000,
        ErrorMessage = "Value for {0} must be between {1} and {2}.")]
    public required int? Scale { get; set; }

    [Required]
    public required int? VerbosityLevel { get; set; }
}

```

In the preceding `SettingsOptions` class, the `ConfigurationSectionName` property contains the name of the configuration section to bind to. In this scenario, the options object provides the name of its configuration section. The use of the following data annotation attributes are used:

- **RequiredAttribute**: Specifies that the property is required.
- **RegularExpressionAttribute**: Specifies that the property value must match the specified regular expression pattern.
- **RangeAttribute**: Specifies that the property value must be within a specified range.

Tip

In addition to the `RequiredAttribute`, the properties also use the `required` modifier. This helps to ensure that consumers of the options object don't forget to set the property value, although it doesn't relate to the validation source generation feature.

The following code exemplifies how to bind the configuration section to the options object and validate the data annotations:

C#

```

using ConsoleJson.Example;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
using Microsoft.Extensions.Options;

HostApplicationBuilder builder = Host.CreateApplicationBuilder(args);

```

```
builder.Services
    .AddOptions<SettingsOptions>()

    .Bind(builder.Configuration.GetSection(SettingsOptions.ConfigurationSectionName));

builder.Services
    .AddSingleton<IValidateOptions<SettingsOptions>,
    ValidateSettingsOptions>();

using IHost app = builder.Build();

var settingsOptions =
    app.Services.GetRequiredService<IOptions<SettingsOptions>>().Value;

await app.RunAsync();
```

By leveraging compile-time source generation for options validation, you can generate performance-optimized validation code and eliminate the need for reflection, resulting in smoother AOT-compatible app building. The following code demonstrates how to use the options validation source generator:

C#

```
using Microsoft.Extensions.Options;

namespace ConsoleJson.Example;

[OptionsValidator]
public partial class ValidateSettingsOptions :
    IValidateOptions<SettingsOptions>
{}
```

The presence of the [OptionsValidatorAttribute](#) on an empty partial class instructs the options validation source generator to create the `IValidateOptions` interface implementation that validates `SettingsOptions`. The code that's generated by the options validation source generator will resemble the following example:

C#

```
// <auto-generated/>
#pragma warning disable CS1591 // Compensate for
https://github.com/dotnet/roslyn/issues/54103
namespace ConsoleJson.Example
{
    partial class ValidateSettingsOptions
```

```

{
    /// <summary>
    /// Validates a specific named options instance (or all when
<paramref name="name"/> is <see langword="null" />).
    /// </summary>
    /// <param name="name">The name of the options instance being
validated.</param>
    /// <param name="options">The options instance.</param>
    /// <returns>Validation result.</returns>

[global::System.CodeDom.Compiler.GeneratedCodeAttribute("Microsoft.Extensions.Options.SourceGeneration", "8.0.9.3103")]

[System.Diagnostics.CodeAnalysis.UnconditionalSuppressMessage("Trimming",
"IL2026:RequiresUnreferencedCode",
Justification = "The created ValidationContext object is used
in a way that never call reflection")]
    public global::Microsoft.Extensions.Options.ValidateOptionsResult
Validate(string? name, global::ConsoleJson.Example.SettingsOptions options)
{
    global::Microsoft.Extensions.Options.ValidateOptionsResultBuilder? builder =
null;
    var context = new
global::System.ComponentModel.DataAnnotations.ValidationContext(options);
    var validationResults = new
global::System.Collections.Generic.List<global::System.ComponentModel.DataAnnotations.ValidationResult>();
    var validationAttributes = new
global::System.Collections.Generic.List<global::System.ComponentModel.DataAnnotations.ValidationAttribute>(2);

        context.MemberName = "SiteTitle";
        context.DisplayName = string.IsNullOrEmpty(name) ?
"SettingsOptions.SiteTitle" : $"{name}.SiteTitle";

validationAttributes.Add(global::__OptionValidationStaticInstances.__Attributes.A1);

validationAttributes.Add(global::__OptionValidationStaticInstances.__Attributes.A2);
    if
(!global::System.ComponentModel.DataAnnotations.Validator.TryValidateValue(o
ptions.SiteTitle, context, validationResults, validationAttributes))
    {
        (builder ??= new()).AddResults(validationResults);
    }

        context.MemberName = "Scale";
        context.DisplayName = string.IsNullOrEmpty(name) ?
"SettingsOptions.Scale" : $"{name}.Scale";
        validationResults.Clear();
        validationAttributes.Clear();

validationAttributes.Add(global::__OptionValidationStaticInstances.__Attribu

```

```

tes.A1);

validationAttributes.Add(global::__OptionValidationStaticInstances.__Attributes.A3);
    if
(!global::System.ComponentModel.DataAnnotations.Validator.TryValidateValue(o
ptions.Scale, context, validationResults, validationAttributes))
    {
        (builder ??= new()).AddResults(validationResults);
    }

        context.MemberName = "VerbosityLevel";
        context.DisplayName = string.IsNullOrEmpty(name) ?
"SettingsOptions.VerbosityLevel" : $"{name}.VerbosityLevel";
        validationResults.Clear();
        validationAttributes.Clear();

validationAttributes.Add(global::__OptionValidationStaticInstances.__Attribu
tes.A1);
    if
(!global::System.ComponentModel.DataAnnotations.Validator.TryValidateValue(o
ptions.VerbosityLevel, context, validationResults, validationAttributes))
    {
        (builder ??= new()).AddResults(validationResults);
    }

    return builder is null ?
global::Microsoft.Extensions.Options.ValidateOptionsResult.Success :
builder.Build();
}
}

namespace __OptionValidationStaticInstances
{

[global::System.CodeDom.Compiler.GeneratedCodeAttribute("Microsoft.Extension
s.Options.SourceGeneration", "8.0.9.3103")]
    file static class __Attributes
    {
        internal static readonly
global::System.ComponentModel.DataAnnotations.RequiredAttribute A1 = new
global::System.ComponentModel.DataAnnotations.RequiredAttribute();

        internal static readonly
global::System.ComponentModel.DataAnnotations.RegularExpressionAttribute A2
= new
global::System.ComponentModel.DataAnnotations.RegularExpressionAttribute(
    "^[a-zA-Z'-'\\s]{1,40}$");

        internal static readonly
__OptionValidationGeneratedAttributes.__SourceGen__RangeAttribute A3 = new
__OptionValidationGeneratedAttributes.__SourceGen__RangeAttribute(
    (int)0,
    (int)1000)
{
}
}

```

```

        ErrorMessage = "Value for {0} must be between {1} and {2}."}
    };
}
}

namespace __OptionValidationStaticInstances
{

[global::System.CodeDom.Compiler.GeneratedCodeAttribute("Microsoft.Extensions.Options.SourceGeneration", "8.0.9.3103")]
    file static class __Validators
    {
    }
}

namespace __OptionValidationGeneratedAttributes
{



[global::System.CodeDom.Compiler.GeneratedCodeAttribute("Microsoft.Extensions.Options.SourceGeneration", "8.0.9.3103")]
    [global::System.AttributeUsage(global::System.AttributeTargets.Property | global::System.AttributeTargets.Field | global::System.AttributeTargets.Parameter, AllowMultiple = false)]
    file class __SourceGen__RangeAttribute :
global::System.ComponentModel.DataAnnotations.ValidationAttribute
{
    public __SourceGen__RangeAttribute(int minimum, int maximum) : base()
    {
        Minimum = minimum;
        Maximum = maximum;
        OperandType = typeof(int);
    }
    public __SourceGen__RangeAttribute(double minimum, double maximum) : base()
    {
        Minimum = minimum;
        Maximum = maximum;
        OperandType = typeof(double);
    }
    public __SourceGen__RangeAttribute(global::System.Type type, string minimum, string maximum) : base()
    {
        OperandType = type;
        NeedToConvertMinMax = true;
        Minimum = minimum;
        Maximum = maximum;
    }
    public object Minimum { get; private set; }
    public object Maximum { get; private set; }
    public bool MinimumIsExclusive { get; set; }
    public bool MaximumIsExclusive { get; set; }
    public global::System.Type OperandType { get; }
    public bool ParseLimitsInInvariantCulture { get; set; }
    public bool ConvertValueInInvariantCulture { get; set; }
    public override string FormatErrorMessage(string name) =>
}
}

```

```

string.Format(global::System.Globalization.CultureInfo.CurrentCulture,
GetValidationErrorMessage(), name, Minimum, Maximum);
    private bool NeedToConvertMinMax { get; }
    private bool Initialized { get; set; }
    public override bool IsValid(object? value)
    {
        if (!Initialized)
        {
            if (Minimum is null || Maximum is null)
            {
                throw new global::System.InvalidOperationException("The
minimum and maximum values must be set to valid values.");
            }
            if (NeedToConvertMinMax)
            {
                System.Globalization.CultureInfo culture =
ParseLimitsInInvariantCulture ?
global::System.Globalization.CultureInfo.InvariantCulture :
global::System.Globalization.CultureInfo.CurrentCulture;
                Minimum = ConvertValue(Minimum, culture) ?? throw new
global::System.InvalidOperationException("The minimum and maximum values
must be set to valid values.");
                Maximum = ConvertValue(Maximum, culture) ?? throw new
global::System.InvalidOperationException("The minimum and maximum values
must be set to valid values.");
            }
            int cmp =
((global::System.IComparable)Minimum).CompareTo((global::System.IComparable)
Maximum);
            if (cmp > 0)
            {
                throw new global::System.InvalidOperationException("The
maximum value '{Maximum}' must be greater than or equal to the minimum value
'{Minimum}'.");
            }
            else if (cmp == 0 && (MinimumIsExclusive ||
MaximumIsExclusive))
            {
                throw new
global::System.InvalidOperationException("Cannot use exclusive bounds when
the maximum value is equal to the minimum value.");
            }
            Initialized = true;
        }

        if (value is null or string { Length: 0 })
        {
            return true;
        }

        System.Globalization.CultureInfo formatProvider =
ConvertValueInInvariantCulture ?
global::System.Globalization.CultureInfo.InvariantCulture :
global::System.Globalization.CultureInfo.CurrentCulture;
        object? convertedValue;
    }
}

```

```

        try
    {
        convertedValue = ConvertValue(value, formatProvider);
    }
    catch (global::System.Exception e) when (e is
global::System.FormatException or global::System.InvalidCastException or
global::System.NotSupportedException)
    {
        return false;
    }

    var min = (global::System.IComparable)Minimum;
    var max = (global::System.IComparable)Maximum;

    return
        (MinimumIsExclusive ? min.CompareTo(convertedValue) < 0 :
min.CompareTo(convertedValue) <= 0) &&
        (MaximumIsExclusive ? max.CompareTo(convertedValue) > 0 :
max.CompareTo(convertedValue) >= 0);
    }

    private string GetValidationErrorMessage()
    {
        return (MinimumIsExclusive, MaximumIsExclusive) switch
        {
            (false, false) => "The field {0} must be between {1} and
{2}.",
            (true, false) => "The field {0} must be between {1}
exclusive and {2}.",
            (false, true) => "The field {0} must be between {1} and {2}
exclusive.",
            (true, true) => "The field {0} must be between {1} exclusive
and {2} exclusive.",
        };
    }

    private object? ConvertValue(object? value,
System.Globalization.CultureInfo formatProvider)
    {
        if (value is string stringValue)
        {
            value = global::System.Convert.ChangeType(stringValue,
OperandType, formatProvider);
        }
        else
        {
            value = global::System.Convert.ChangeType(value,
OperandType, formatProvider);
        }
        return value;
    }
}

```

The generated code is optimized for performance and doesn't rely on reflection. It's also AOT-compatible. The generated code is placed in a file named `Validators.g.cs`.

ⓘ Note

You don't need to take any additional steps to enable the options validation source generator. It's automatically enabled by default when your project references `Microsoft.Extensions.Options` [↗] version 8 or later, or when building an ASP.NET application.

The only step you need to take is to add the following to the startup code:

C#

```
builder.Services
    .AddSingleton<IValidateOptions<SettingsOptions>,
    ValidateSettingsOptions>();
```

ⓘ Note

Calling `OptionsBuilderDataAnnotationsExtensions.ValidateDataAnnotations<TOptions>(OptionsBuilder<TOptions>)` isn't required when using the options validation source generator.

When the application attempts to access the options object, the generated code for options validation is executed to validate the options object. The following code snippet illustrates how to access the options object:

C#

```
var settingsOptions =
    app.Services.GetRequiredService<IOptions<SettingsOptions>>().Value;
```

Replaced data annotation attributes

Upon close examination of the generated code, you'll observe that the original data annotation attributes, such as `RangeAttribute`, that were initially applied to the property `SettingsOptions.Scale`, have been substituted with custom attributes like `__SourceGen__RangeAttribute`. This substitution is made because the `RangeAttribute`

relies on reflection for validation. In contrast, `_SourceGen_RangeAttribute` is a custom attribute optimized for performance and doesn't depend on reflection, making the code AOT-compatible. The same pattern of attribute replacement will be applied on [MaxLengthAttribute](#), [MinLengthAttribute](#), and [LengthAttribute](#) in addition to [RangeAttribute](#).

For anyone developing custom data annotation attributes, it's advisable to refrain from using reflection for validation. Instead, it's recommended to craft strongly typed code that doesn't rely on reflection. This approach ensures smooth compatibility with AOT builds.

See also

[Options pattern](#)

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

.NET is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

Options pattern guidance for .NET library authors

Article • 06/23/2023

With the help of dependency injection, registering your services and their corresponding configurations can make use of the *options pattern*. The options pattern enables consumers of your library (and your services) to require instances of [options interfaces](#) where `TOptions` is your options class. Consuming configuration options through strongly-typed objects helps to ensure consistent value representation, enables validation with data annotations, and removes the burden of manually parsing string values. There are many [configuration providers](#) for consumers of your library to use. With these providers, consumers can configure your library in many ways.

As a .NET library author, you'll learn general guidance on how to correctly expose the options pattern to consumers of your library. There are various ways to achieve the same thing, and several considerations to make.

Naming conventions

By convention, extension methods responsible for registering services are named `Add{Service}`, where `{Service}` is a meaningful and descriptive name. `Add{Service}` extension methods are commonplace in [ASP.NET Core](#) and .NET alike.

- ✓ CONSIDER names that disambiguate your service from other offerings.
- ✗ DO NOT use names that are already part of the .NET ecosystem from official Microsoft packages.
- ✓ CONSIDER naming static classes that expose extension methods as `{Type}Extensions`, where `{Type}` is the type that you're extending.

Namespace guidance

Microsoft packages make use of the `Microsoft.Extensions.DependencyInjection` namespace to unify the registration of various service offerings.

- ✓ CONSIDER a namespace that clearly identifies your package offering.
- ✗ DO NOT use the `Microsoft.Extensions.DependencyInjection` namespace for non-official Microsoft packages.

Parameterless

If your service can work with minimal or no explicit configuration, consider a parameterless extension method.

C#

```
using Microsoft.Extensions.DependencyInjection;

namespace ExampleLibrary.Extensions.DependencyInjection;

public static class ServiceCollectionExtensions
{
    public static IServiceCollection AddMyLibraryService(
        this IServiceCollection services)
    {
        services.AddOptions<LibraryOptions>()
            .Configure(options =>
        {
            // Specify default option values
        });

        // Register lib services here...
        // services.AddScoped<ILibraryService, DefaultLibraryService>();

        return services;
    }
}
```

In the preceding code, the `AddMyLibraryService`:

- Extends an instance of `IServiceCollection`
- Calls `OptionsServiceCollectionExtensions.AddOptions<TOptions>(IServiceCollection)` with the type parameter of `LibraryOptions`
- Chains a call to `Configure`, which specifies the default option values

IConfiguration parameter

When you author a library that exposes many options to consumers, you may want to consider requiring an `IConfiguration` parameter extension method. The expected `IConfiguration` instance should be scoped to a named section of the configuration by using the `IConfiguration.GetSection` function.

C#

```
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;
```

```
namespace ExampleLibrary.Extensions.DependencyInjection;

public static class ServiceCollectionExtensions
{
    public static IServiceCollection AddMyLibraryService(
        this IServiceCollection services,
        IConfiguration namedConfigurationSection)
    {
        // Default library options are overridden
        // by bound configuration values.
        services.Configure<LibraryOptions>(namedConfigurationSection);

        // Register lib services here...
        // services.AddScoped<ILibraryService, DefaultLibraryService>();

        return services;
    }
}
```

💡 Tip

The `Configure<TOptions>(IServiceCollection, IConfiguration)` method is part of the [Microsoft.Extensions.Options.ConfigurationExtensions](#) NuGet package.

In the preceding code, the `AddMyLibraryService`:

- Extends an instance of `IServiceCollection`
- Defines an `IConfiguration` parameter `namedConfigurationSection`
- Calls `Configure<TOptions>(IServiceCollection, IConfiguration)` passing the generic type parameter of `LibraryOptions` and the `namedConfigurationSection` instance to configure

Consumers in this pattern provide the scoped `IConfiguration` instance of the named section:

C#

```
using ExampleLibrary.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;

HostApplicationBuilder builder = Host.CreateApplicationBuilder(args);

builder.Services.AddMyLibraryService(
    builder.Configuration.GetSection("LibraryOptions"));

using IHost host = builder.Build();
```

```
// Application code should start here.  
  
await host.RunAsync();
```

The call to `.AddMyLibraryService` is made on the `IServiceCollection` type.

As the library author, specifying default values is up to you.

ⓘ Note

It is possible to bind configuration to an options instance. However, there is a risk of name collisions - which will cause errors. Additionally, when manually binding in this way, you limit the consumption of your options pattern to read-once. Changes to settings will not be re-bound, as such consumers will not be able to use the `IOptionsMonitor` interface.

C#

```
services.AddOptions<LibraryOptions>()  
    .Configure< IConfiguration>(  
        (options, configuration) =>  
            configuration.GetSection("LibraryOptions").Bind(options));
```

Instead, you should use the `BindConfiguration` extension method. This extension method binds the configuration to the options instance, and also registers a change token source for the configuration section. This allows consumers to use the `IOptionsMonitor` interface.

Configuration section path parameter

Consumers of your library may want to specify the configuration section path to bind your underlying `IOptions` type. In this scenario, you define a `string` parameter in your extension method.

C#

```
using Microsoft.Extensions.DependencyInjection;  
  
namespace ExampleLibrary.Extensions.DependencyInjection;  
  
public static class ServiceCollectionExtensions  
{  
    public static IServiceCollection AddMyLibraryService(  
        this IServiceCollection services,
```

```

        string configSectionPath)
    {
        services.AddOptions<SupportOptions>()
            .BindConfiguration(configSectionPath)
            .ValidateDataAnnotations()
            .ValidateOnStart();

        // Register lib services here...
        // services.AddScoped<ILibraryService, DefaultLibraryService>();

        return services;
    }
}

```

In the preceding code, the `AddMyLibraryService`:

- Extends an instance of `IServiceCollection`
- Defines a `string` parameter `configSectionPath`
- Calls:
 - `AddOptions` with the generic type parameter of `SupportOptions`
 - `BindConfiguration` with the given `configSectionPath` parameter
 - `ValidateDataAnnotations` to enable data annotation validation
 - `ValidateOnStart` to enforce validation on start rather than in runtime

In the next example, the [Microsoft.Extensions.Options.DataAnnotations](#) NuGet package is used to enable data annotation validation. The `SupportOptions` class is defined as follows:

C#

```

using System.ComponentModel.DataAnnotations;

public sealed class SupportOptions
{
    [Url]
    public string? Url { get; set; }

    [Required, EmailAddress]
    public required string Email { get; set; }

    [Required, DataType(DataType.PhoneNumber)]
    public required string PhoneNumber { get; set; }
}

```

Imagine that the following JSON `appsettings.json` file is used:

JavaScript

```
{  
    "Support": {  
        "Url": "https://support.example.com",  
        "Email": "help@support.example.com",  
        "PhoneNumber": "+1(888)-SUPPORT"  
    }  
}
```

Action<TOptions> parameter

Consumers of your library may be interested in providing a lambda expression that yields an instance of your options class. In this scenario, you define an `Action<LibraryOptions>` parameter in your extension method.

C#

```
using Microsoft.Extensions.DependencyInjection;  
  
namespace ExampleLibrary.Extensions.DependencyInjection;  
  
public static class ServiceCollectionExtensions  
{  
    public static IServiceCollection AddMyLibraryService(  
        this IServiceCollection services,  
        Action<LibraryOptions> configureOptions)  
    {  
        services.Configure(configureOptions);  
  
        // Register lib services here...  
        // services.AddScoped<ILibraryService, DefaultLibraryService>();  
  
        return services;  
    }  
}
```

In the preceding code, the `AddMyLibraryService`:

- Extends an instance of `IServiceCollection`
- Defines an `Action<T>` parameter `configureOptions` where `T` is `LibraryOptions`
- Calls `Configure` given the `configureOptions` action

Consumers in this pattern provide a lambda expression (or a delegate that satisfies the `Action<LibraryOptions>` parameter):

C#

```
using ExampleLibrary.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;

HostApplicationBuilder builder = Host.CreateApplicationBuilder(args);

builder.Services.AddMyLibraryService(options =>
{
    // User defined option values
    // options.SomePropertyValue = ...
});

using IHost host = builder.Build();

// Application code should start here.

await host.RunAsync();
```

Options instance parameter

Consumers of your library might prefer to provide an inlined options instance. In this scenario, you expose an extension method that takes an instance of your options object, `LibraryOptions`.

C#

```
using Microsoft.Extensions.DependencyInjection;

namespace ExampleLibrary.Extensions.DependencyInjection;

public static class ServiceCollectionExtensions
{
    public static IServiceCollection AddMyLibraryService(
        this IServiceCollection services,
        LibraryOptions userOptions)
    {
        services.AddOptions<LibraryOptions>()
            .Configure(options =>
            {
                // Overwrite default option values
                // with the user provided options.
                // options.SomeValue = userOptions.SomeValue;
            });

        // Register lib services here...
        // services.AddScoped<ILibraryService, DefaultLibraryService>();

        return services;
    }
}
```

In the preceding code, the `AddMyLibraryService`:

- Extends an instance of `IServiceCollection`
- Calls `OptionsServiceCollectionExtensions.AddOptions<TOptions>(IServiceCollection)` with the type parameter of `LibraryOptions`
- Chains a call to `Configure`, which specifies default option values that can be overridden from the given `userOptions` instance

Consumers in this pattern provide an instance of the `LibraryOptions` class, defining desired property values inline:

C#

```
using ExampleLibrary.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;

HostApplicationBuilder builder = Host.CreateApplicationBuilder(args);

builder.Services.AddMyLibraryService(new LibraryOptions
{
    // Specify option values
    // SomePropertyValue = ...
});

using IHost host = builder.Build();

// Application code should start here.

await host.RunAsync();
```

Post configuration

After all configuration option values are bound or specified, post configuration functionality is available. Exposing the same `Action<TOptions>` parameter detailed earlier, you could choose to call `PostConfigure`. Post configure runs after all `.Configure` calls. There are few reasons why you'd want to consider using `PostConfigure`:

- **Execution order:** You can override any configuration values that were set in the `.Configure` calls.
- **Validation:** You can validate the default values have been set after all other configurations have been applied.

C#

```
using Microsoft.Extensions.DependencyInjection;
```

```
namespace ExampleLibrary.Extensions.DependencyInjection;

public static class ServiceCollectionExtensions
{
    public static IServiceCollection AddMyLibraryService(
        this IServiceCollection services,
        Action<LibraryOptions> configureOptions)
    {
        services.PostConfigure(configureOptions);

        // Register lib services here...
        // services.AddScoped<ILibraryService, DefaultLibraryService>();

        return services;
    }
}
```

In the preceding code, the `AddMyLibraryService`:

- Extends an instance of `IServiceCollection`
- Defines an `Action<T>` parameter `configureOptions` where `T` is `LibraryOptions`
- Calls `PostConfigure` given the `configureOptions` action

Consumers in this pattern provide a lambda expression (or a delegate that satisfies the `Action<LibraryOptions>` parameter), just as they would with the `Action<TOptions>` parameter in a non-post configuration scenario:

C#

```
using ExampleLibrary.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;

HostApplicationBuilder builder = Host.CreateApplicationBuilder(args);

builder.Services.AddMyLibraryService(options =>
{
    // Specify option values
    // options.SomePropertyValue = ...
});

using IHost host = builder.Build();

// Application code should start here.

await host.RunAsync();
```

See also

- Options pattern in .NET
- Dependency injection in .NET
- Dependency injection guidelines

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

Logging in C# and .NET

Article • 12/11/2023

.NET supports high performance, structured logging via the [ILogger](#) API to help monitor application behavior and diagnose issues. Logs can be written to different destinations by configuring different [logging providers](#). Basic logging providers are built-in and there are many third-party providers available as well.

Get started

This first example shows the basics, but it is only suitable for a trivial console app. In the next section you see how to improve the code considering scale, performance, configuration and typical programming patterns.

C#

```
using Microsoft.Extensions.Logging;

using ILoggerFactory factory = LoggerFactory.Create(builder =>
builder.AddConsole());
ILogger logger = factory.CreateLogger("Program");
logger.LogInformation("Hello World! Logging is {Description}.", "fun");
```

The preceding example:

- Creates an [ILoggerFactory](#). The `ILoggerFactory` stores all the configuration that determines where log messages are sent. In this case, you configure the console [logging provider](#) so that log messages are written to the console.
- Creates an [ILogger](#) with a category named "Program". The `category` is a `string` that is associated with each message logged by the `ILogger` object. It's used to group log messages from the same class (or category) together when searching or filtering logs.
- Calls [.LogInformation](#) to log a message at the `Information` level. The [log level](#) indicates the severity of the logged event and is used to filter out less important log messages. The log entry also includes a [message template](#) `"Hello World! Logging is {Description}."` and a key-value pair `Description = fun`. The key name (or placeholder) comes from the word inside the curly braces in the template and the value comes from the remaining method argument.

💡 Tip

All of the logging example source code is available in the [Samples Browser](#) for download. For more information, see [Browse code samples: Logging in .NET](#).

Logging in a non-trivial app

There are several changes you should consider making to the previous example when logging in a less trivial scenario:

- If your application is using [Dependency Injection \(DI\)](#) or a host such as ASP.NET's [WebApplication](#) or [Generic Host](#) then you should use `ILoggerFactory` and `ILogger` objects from their respective DI containers rather than creating them directly. For more information, see [Integration with DI and Hosts](#).
- Logging [compile-time source generation](#) is usually a better alternative to `ILogger` extension methods like `LogInformation`. Logging source generation offers better performance, stronger typing, and avoids spreading `string` constants throughout your methods. The tradeoff is that using this technique requires a bit more code.

C#

```
using Microsoft.Extensions.Logging;

internal partial class Program
{
    static void Main(string[] args)
    {
        using ILoggerFactory factory = LoggerFactory.Create(builder =>
builder.AddConsole());
        ILogger logger = factory.CreateLogger("Program");
        LogStartupMessage(logger, "fun");
    }

    [LoggerMessage(Level = LogLevel.Information, Message = "Hello World!
Logging is {Description}.")]
    static partial void LogStartupMessage	ILogger logger, string
description);
}
```

- The recommended practice for log category names is to use the fully qualified name of the class that's creating the log message. This helps relate log messages back to the code which produced them and offers a good level of control when filtering logs. `CreateLogger` accepts a `Type` to make this naming easy to do.

C#

```
using Microsoft.Extensions.Logging;

internal class Program
{
    static void Main(string[] args)
    {
        using ILoggerFactory factory = LoggerFactory.Create(builder =>
builder.AddConsole());
        ILogger logger = factory.CreateLogger<Program>();
        logger.LogInformation("Hello World! Logging is {Description}.",
"fun");
    }
}
```

- If you don't use console logs as your sole production monitoring solution, add the [logging providers](#) you plan to use. For example, you could use [OpenTelemetry](#) to send logs over [OTLP \(OpenTelemetry protocol\)](#):

C#

```
using Microsoft.Extensions.Logging;
using OpenTelemetry.Logs;

using ILoggerFactory factory = LoggerFactory.Create(builder =>
{
    builder.AddOpenTelemetry(logging =>
    {
        logging.AddOtlpExporter();
    });
});
ILogger logger = factory.CreateLogger("Program");
logger.LogInformation("Hello World! Logging is {Description}.", "fun");
```

Integration with hosts and dependency injection

If your application is using [Dependency Injection \(DI\)](#) or a host such as ASP.NET's [WebApplication](#) or [Generic Host](#) then you should use `ILoggerFactory` and `ILogger` objects from the DI container rather than creating them directly.

Get an `ILogger` from DI

This example gets an `ILogger` object in a hosted app using [ASP.NET Minimal APIs](#):

C#

```

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddSingleton<ExampleHandler>();

var app = builder.Build();

var handler = app.Services.GetRequiredService<ExampleHandler>();
app.MapGet("/", handler.HandleRequest);

app.Run();

partial class ExampleHandler(ILogger<ExampleHandler> logger)
{
    public string HandleRequest()
    {
        LogHandleRequest(logger);
        return "Hello World";
    }

    [LoggerMessage(LogLevel.Information, "ExampleHandler.HandleRequest was
called")]
    public static partial void LogHandleRequest(ILogger logger);
}

```

The preceding example:

- Created a singleton service called `ExampleHandler` and mapped incoming web requests to run the `ExampleHandler.HandleRequest` function.
- Line 8 defines a [primary constructor](#) for the `ExampleHandler`, a feature added in C# 12. Using the older style C# constructor would work equally well but is a little more verbose.
- The constructor defines a parameter of type `ILogger<ExampleHandler>`. `ILogger<TCategoriesName>` derives from `ILogger` and indicates which category the `ILogger` object has. The DI container locates an `ILogger` with the correct category and supplies it as the constructor argument. If no `ILogger` with that category exists yet, the DI container automatically creates it from the `ILoggerFactory` in the service provider.
- The `logger` parameter received in the constructor was used for logging in the `HandleRequest` function.

Host-provided `ILoggerFactory`

Host builders initialize [default configuration](#), then add a configured `ILoggerFactory` object to the host's DI container when the host is built. Before the host is built you can adjust the logging configuration via [HostApplicationBuilder.Logging](#),

`WebApplicationBuilder.Logging`, or similar APIs on other hosts. Hosts also apply logging configuration from default configuration sources as `appsettings.json` and environment variables. For more information, see [Configuration in .NET](#).

This example expands on the previous one to customize the `ILoggerFactory` provided by `WebApplicationBuilder`. It adds [OpenTelemetry](#) as a logging provider transmitting the logs over [OTLP \(OpenTelemetry protocol\)](#):

C#

```
var builder = WebApplication.CreateBuilder(args);
builder.Logging.AddOpenTelemetry(logging => logging.AddOtlpExporter());
builder.Services.AddSingleton<ExampleHandler>();
var app = builder.Build();
```

Create an `ILoggerFactory` with DI

If you're using a DI container without a host, use [AddLogging](#) to configure and add `ILoggerFactory` to the container.

C#

```
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Logging;

// Add services to the container including logging
var services = new ServiceCollection();
services.AddLogging(builder => builder.AddConsole());
services.AddSingleton<ExampleService>();
IServiceProvider serviceProvider = services.BuildServiceProvider();

// Get the ExampleService object from the container
ExampleService service = serviceProvider.GetRequiredService<ExampleService>();

// Do some pretend work
service.DoSomeWork(10, 20);

class ExampleService(ILogger<ExampleService> logger)
{
    public void DoSomeWork(int x, int y)
    {
        logger.LogInformation("DoSomeWork was called. x={X}, y={Y}", x, y);
    }
}
```

The preceding example:

- Created a DI service container containing an `ILoggerFactory` configured to write to the console
- Added a singleton `ExampleService` to the container
- Created an instance of the `ExampleService` from the DI container which also automatically created an `ILogger<ExampleService>` to use as the constructor argument.
- Invoked `ExampleService.DoSomeWork` which used the `ILogger<ExampleService>` to log a message to the console.

Configure logging

Logging configuration is set in code or via external sources, such as, config files and environment variables. Using external configuration is beneficial when possible because it can be changed without rebuilding the application. However, some tasks, such as setting logging providers, can only be configured from code.

Configure logging without code

For apps that [use a host](#), logging configuration is commonly provided by the `"Logging"` section of `appsettings.{Environment}.json` files. For apps that don't use a host, external configuration sources are [set up explicitly](#) or [configured in code](#) instead.

The following `appsettings.Development.json` file is generated by the .NET Worker service templates:

```
JSON

{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft": "Warning",
      "Microsoft.Hosting.Lifetime": "Information"
    }
  }
}
```

In the preceding JSON:

- The `"Default"`, `"Microsoft"`, and `"Microsoft.Hosting.Lifetime"` log level categories are specified.

- The "Default" value is applied to all categories that aren't otherwise specified, effectively making all default values for all categories "Information". You can override this behavior by specifying a value for a category.
- The "Microsoft" category applies to all categories that start with "Microsoft".
- The "Microsoft" category logs at a log level of Warning and higher.
- The "Microsoft.Hosting.Lifetime" category is more specific than the "Microsoft" category, so the "Microsoft.Hosting.Lifetime" category logs at log level "Information" and higher.
- A specific log provider is not specified, so LogLevel applies to all the enabled logging providers except for the [Windows EventLog](#).

The Logging property can have LogLevel and log provider properties. The LogLevel specifies the minimum level to log for selected categories. In the preceding JSON, Information and Warning log levels are specified. LogLevel indicates the severity of the log and ranges from 0 to 6:

Trace = 0, Debug = 1, Information = 2, Warning = 3, Error = 4, Critical = 5, and None = 6.

When a LogLevel is specified, logging is enabled for messages at the specified level and higher. In the preceding JSON, the Default category is logged for Information and higher. For example, Information, Warning, Error, and Critical messages are logged. If no LogLevel is specified, logging defaults to the Information level. For more information, see [Log levels](#).

A provider property can specify a LogLevel property. LogLevel under a provider specifies levels to log for that provider, and overrides the non-provider log settings. Consider the following *appsettings.json* file:

```
JSON
{
  "Logging": {
    "LogLevel": {
      "Default": "Error",
      "Microsoft": "Warning"
    },
    "Debug": {
      "LogLevel": {
        "Default": "Information",
        "Microsoft.Hosting": "Trace"
      }
    },
    "EventSource": {
      "LogLevel": {
        "Default": "Information"
      }
    }
  }
}
```

```
        "Default": "Warning"
    }
}
}
```

Settings in `Logging.{ProviderName}.LogLevel` override settings in `Logging.LogLevel`. In the preceding JSON, the `Debug` provider's default log level is set to `Information`:

```
Logging:Debug:LogLevel:Default:Information
```

The preceding setting specifies the `Information` log level for every `Logging:Debug:` category except `Microsoft.Hosting`. When a specific category is listed, the specific category overrides the default category. In the preceding JSON, the `Logging:Debug:LogLevel` categories `"Microsoft.Hosting"` and `"Default"` override the settings in `Logging:LogLevel`.

The minimum log level can be specified for any of:

- Specific providers: For example,
`Logging:EventSource:LogLevel:Default:Information`
- Specific categories: For example, `Logging:LogLevel:Microsoft:Warning`
- All providers and all categories: `Logging:LogLevel:Default:Warning`

Any logs below the minimum level are *not*:

- Passed to the provider.
- Logged or displayed.

To suppress all logs, specify `LogLevel.None`. `LogLevel.None` has a value of 6, which is higher than `LogLevel.Critical` (5).

If a provider supports [log scopes](#), `IncludeScopes` indicates whether they're enabled. For more information, see [log scopes](#)

The following `appsettings.json` file contains settings for all of the built-in providers:

JSON

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Error",
      "Microsoft": "Warning",
      "Microsoft.Hosting.Lifetime": "Warning"
    },
  }
```

```

    "Debug": {
        "LogLevel": {
            "Default": "Information"
        }
    },
    "Console": {
        "IncludeScopes": true,
        "LogLevel": {
            "Microsoft.Extensions.Hosting": "Warning",
            "Default": "Information"
        }
    },
    "EventSource": {
        "LogLevel": {
            "Microsoft": "Information"
        }
    },
    "EventLog": {
        "LogLevel": {
            "Microsoft": "Information"
        }
    },
    "AzureAppServicesFile": {
        "IncludeScopes": true,
        "LogLevel": {
            "Default": "Warning"
        }
    },
    "AzureAppServicesBlob": {
        "IncludeScopes": true,
        "LogLevel": {
            "Microsoft": "Information"
        }
    },
    "ApplicationInsights": {
        "LogLevel": {
            "Default": "Information"
        }
    }
}
}

```

In the preceding sample:

- The categories and levels are not suggested values. The sample is provided to show all the default providers.
- Settings in `Logging.{ProviderName}.LogLevel` override settings in `Logging.LogLevel`. For example, the level in `Debug(LogLevel.Default)` overrides the level in `LogLevel.Default`.
- Each provider's *alias* is used. Each provider defines an *alias* that can be used in configuration in place of the fully qualified type name. The built-in providers'

aliases are:

- `Console`
- `Debug`
- `EventSource`
- `EventLog`
- `AzureAppServicesFile`
- `AzureAppServicesBlob`
- `ApplicationInsights`

Set log level by command line, environment variables, and other configuration

Log level can be set by any of the [configuration providers](#). For example, you can create a persisted environment variable named `Logging:LogLevel:Microsoft` with a value of `Information`.

Command Line

Create and assign persisted environment variable, given the log level value.

CMD

```
:: Assigns the env var to the value  
setx "Logging__LogLevel__Microsoft" "Information" /M
```

In a *new* instance of the **Command Prompt**, read the environment variable.

CMD

```
:: Prints the env var value  
echo %Logging__LogLevel__Microsoft%
```

The preceding environment setting is persisted in the environment. To test the settings when using an app created with the .NET Worker service templates, use the `dotnet run` command in the project directory after the environment variable is assigned.

.NET CLI

```
dotnet run
```

Tip

After setting an environment variable, restart your integrated development environment (IDE) to ensure that newly added environment variables are available.

On [Azure App Service](#), select **New application setting** on the **Settings > Configuration** page. Azure App Service application settings are:

- Encrypted at rest and transmitted over an encrypted channel.
- Exposed as environment variables.

For more information on setting .NET configuration values using environment variables, see [environment variables](#).

Configure logging with code

To configure logging in code, use the [ILoggingBuilder](#) API. This can be accessed from different places:

- When creating the `ILoggerFactory` directly, configure in [LoggerFactory.Create](#).
- When using DI without a host, configure in [LoggingServiceCollectionExtensions.AddLogging](#).
- When using a host, configure with [HostApplicationBuilder.Logging](#), [WebApplicationBuilder.Logging](#) or other host specific APIs.

This example shows setting the console [logging provider](#) and several [filters](#).

C#

```
using Microsoft.Extensions.Logging;

using var loggerFactory = LoggerFactory.Create(static builder =>
{
    builder
        .AddFilter("Microsoft", LogLevel.Warning)
        .AddFilter("System", LogLevel.Warning)
        .AddFilter("LoggingConsoleApp.Program", LogLevel.Debug)
        .AddConsole();
});

	ILogger logger = loggerFactory.CreateLogger<Program>();
	logger.LogDebug("Hello {Target}", "Everyone");
```

In the preceding example `AddFilter` is used to [adjust the log level](#) that's enabled for various categories. `AddConsole` is used to add the console logging provider. By default,

logs with `Debug` severity aren't enabled, but because the configuration adjusted the filters, the debug message "Hello Everyone" is displayed on the console.

How filtering rules are applied

When an `ILogger<TCategories>` object is created, the `ILoggerFactory` object selects a single rule per provider to apply to that logger. All messages written by an `ILogger` instance are filtered based on the selected rules. The most specific rule for each provider and category pair is selected from the available rules.

The following algorithm is used for each provider when an `ILogger` is created for a given category:

- Select all rules that match the provider or its alias. If no match is found, select all rules with an empty provider.
- From the result of the preceding step, select rules with longest matching category prefix. If no match is found, select all rules that don't specify a category.
- If multiple rules are selected, take the **last** one.
- If no rules are selected, use `LoggingBuilderExtensions.SetMinimumLevel(ILoggingBuilder, LogLevel)` to specify the minimum logging level.

Log category

When an `ILogger` object is created, a *category* is specified. That category is included with each log message created by that instance of `ILogger`. The category string is arbitrary, but the convention is to use the class name. For example, in an application with a service defined like the following object, the category might be

`"Example.DefaultService"`:

C#

```
namespace Example
{
    public class DefaultService : IService
    {
        private readonly ILogger<DefaultService> _logger;

        public DefaultService(ILogger<DefaultService> logger) =>
            _logger = logger;

        // ...
    }
}
```

```
    }  
}
```

To explicitly specify the category, call [LoggerFactory.CreateLogger](#):

C#

```
namespace Example  
{  
    public class DefaultService : IService  
    {  
        private readonly ILogger _logger;  
  
        public DefaultService	ILoggerFactory loggerFactory) =>  
            _logger = loggerFactory.CreateLogger("CustomCategory");  
  
        // ...  
    }  
}
```

Calling `CreateLogger` with a fixed name can be useful when used in multiple classes/types so the events can be organized by category.

`ILogger<T>` is equivalent to calling `CreateLogger` with the fully qualified type name of `T`.

Log level

The following table lists the [LogLevel](#) values, the convenience `Log{LogLevel}` extension method, and the suggested usage:

[+] [Expand table](#)

| LogLevel | Value | Method | Description |
|-------------|-------|---------------------------------|---|
| Trace | 0 | LogTrace | Contain the most detailed messages. These messages may contain sensitive app data. These messages are disabled by default and should not be enabled in production. |
| Debug | 1 | LogDebug | For debugging and development. Use with caution in production due to the high volume. |
| Information | 2 | .LogInformation | Tracks the general flow of the app. May have long-term value. |
| Warning | 3 | .LogWarning | For abnormal or unexpected events. Typically includes errors or conditions that don't cause the app to fail. |

| LogLevel | Value | Method | Description |
|----------|-------|-------------|---|
| Error | 4 | LogError | For errors and exceptions that cannot be handled. These messages indicate a failure in the current operation or request, not an app-wide failure. |
| Critical | 5 | LogCritical | For failures that require immediate attention. Examples: data loss scenarios, out of disk space. |
| None | 6 | | Specifies that no messages should be written. |

In the previous table, the `LogLevel` is listed from lowest to highest severity.

The `Log` method's first parameter, `LogLevel`, indicates the severity of the log. Rather than calling `Log(LogLevel, ...)`, most developers call the `Log{LogLevel}` extension methods. The `Log{LogLevel}` extension methods [call the Log method and specify the LogLevel ↗](#). For example, the following two logging calls are functionally equivalent and produce the same log:

C#

```
public void LogDetails()
{
    var logMessage = "Details for log.";

    _logger.Log(LogLevel.Information, AppLogEvents.Details, logMessage);
    _logger.LogInformation(AppLogEvents.Details, logMessage);
}
```

`AppLogEvents.Details` is the event ID, and is implicitly represented by a constant `Int32` value. `AppLogEvents` is a class that exposes various named identifier constants and is displayed in the [Log event ID](#) section.

The following code creates `Information` and `Warning` logs:

C#

```
public async Task<T> GetAsync<T>(string id)
{
    _logger.LogInformation(AppLogEvents.Read, "Reading value for {Id}", id);

    var result = await _repository.GetAsync(id);
    if (result is null)
    {
        _logger.LogWarning(AppLogEvents.ReadNotFound, "GetAsync({Id}) not found", id);
    }
}
```

```
    return result;  
}
```

In the preceding code, the first `Log{LogLevel}` parameter, `AppLogEvents.Read`, is the [Log event ID](#). The second parameter is a message template with placeholders for argument values provided by the remaining method parameters. The method parameters are explained in the [message template](#) section later in this article.

Configure the appropriate log level and call the correct `Log{LogLevel}` methods to control how much log output is written to a particular storage medium. For example:

- In production:
 - Logging at the `Trace` or `Debug` levels produces a high-volume of detailed log messages. To control costs and not exceed data storage limits, log `Trace` and `Debug` level messages to a high-volume, low-cost data store. Consider limiting `Trace` and `Debug` to specific categories.
 - Logging at `Warning` through `Critical` levels should produce few log messages.
 - Costs and storage limits usually aren't a concern.
 - Few logs allow more flexibility in data store choices.
- In development:
 - Set to `Warning`.
 - Add `Trace` or `Debug` messages when troubleshooting. To limit output, set `Trace` or `Debug` only for the categories under investigation.

The following JSON sets `Logging:Console:LogLevel:Microsoft:Information`:

JSON

```
{  
  "Logging": {  
    "LogLevel": {  
      "Microsoft": "Warning"  
    },  
    "Console": {  
      "LogLevel": {  
        "Microsoft": "Information"  
      }  
    }  
  }  
}
```

Log event ID

Each log can specify an *event identifier*, the `EventId` is a structure with an `Id` and optional `Name` readonly properties. The sample source code uses the `AppLogEvents` class to define event IDs:

C#

```
using Microsoft.Extensions.Logging;

internal static class AppLogEvents
{
    internal EventId Create = new(1000, "Created");
    internal EventId Read = new(1001, "Read");
    internal EventId Update = new(1002, "Updated");
    internal EventId Delete = new(1003, "Deleted");

    // These are also valid EventId instances, as there's
    // an implicit conversion from int to an EventId
    internal const int Details = 3000;
    internal const int Error = 3001;

    internal EventId ReadNotFound = 4000;
    internal EventId UpdateNotFound = 4001;

    // ...
}
```

Tip

For more information on converting an `int` to an `EventId`, see [EventId.Implicit\(Int32 to EventId\) Operator](#).

An event ID associates a set of events. For example, all logs related to reading values from a repository might be `1001`.

The logging provider may log the event ID in an ID field, in the logging message, or not at all. The Debug provider doesn't show event IDs. The console provider shows event IDs in brackets after the category:

Console

```
info: Example.DefaultService.GetAsync[1001]
      Reading value for a1b2c3
warn: Example.DefaultService.GetAsync[4000]
      GetAsync(a1b2c3) not found
```

Some logging providers store the event ID in a field, which allows for filtering on the ID.

Log message template

Each log API uses a message template. The message template can contain placeholders for which arguments are provided. Use names for the placeholders, not numbers. The order of placeholders, not their names, determines which parameters are used to provide their values. In the following code, the parameter names are out of sequence in the message template:

```
C#
```

```
string p1 = "param1";
string p2 = "param2";
_logger.LogInformation("Parameter values: {p2}, {p1}", p1, p2);
```

The preceding code creates a log message with the parameter values in sequence:

```
text
```

```
Parameter values: param1, param2
```

ⓘ Note

Be mindful when using multiple placeholders within a single message template, as they're ordinal-based. The names are *not* used to align the arguments to the placeholders.

This approach allows logging providers to implement [semantic or structured logging](#). The arguments themselves are passed to the logging system, not just the formatted message template. This enables logging providers to store the parameter values as fields. Consider the following logger method:

```
C#
```

```
_logger.LogInformation("Getting item {Id} at {RunTime}", id, DateTime.Now);
```

For example, when logging to Azure Table Storage:

- Each Azure Table entity can have `ID` and `RunTime` properties.
- Tables with properties simplify queries on logged data. For example, a query can find all logs within a particular `RunTime` range without having to parse the time out of the text message.

Log message template formatting

Log message templates support placeholder formatting. Templates are free to specify [any valid format](#) for the given type argument. For example, consider the following `Information` logger message template:

C#

```
_logger.LogInformation("Logged on {PlaceHolderName:MMMM dd, yyyy}",  
DateTimeOffset.UtcNow);  
// Logged on January 06, 2022
```

In the preceding example, the `DateTimeOffset` instance is the type that corresponds to the `PlaceHolderName` in the logger message template. This name can be anything as the values are ordinal-based. The `MMMM dd, yyyy` format is valid for the `DateTimeOffset` type.

For more information on `DateTime` and `DateTimeOffset` formatting, see [Custom date and time format strings](#).

Examples

The following examples show how to format a message template using the `{}` placeholder syntax. Additionally, an example of escaping the `{}` placeholder syntax is shown with its output. Finally, string interpolation with templating placeholders is also shown:

C#

```
logger.LogInformation("Number: {Number}", 1); // Number: 1  
logger.LogInformation("{Number}): {Number}", 3); // {Number}: 3  
logger.LogInformation($"{{{Number}}}: {{Number}}", 5); // {Number}: 5
```

Tip

- In most cases, you should use log message template formatting when logging. Use of string interpolation can cause performance issues.
- Code analysis rule [CA2254: Template should be a static expression](#) helps alert you to places where your log messages don't use proper formatting.

Log exceptions

The logger methods have overloads that take an exception parameter:

C#

```
public void Test(string id)
{
    try
    {
        if (id is "none")
        {
            throw new Exception("Default Id detected.");
        }
    }
    catch (Exception ex)
    {
        _logger.LogWarning(
            AppLogEvents.Error, ex,
            "Failed to process iteration: {Id}", id);
    }
}
```

Exception logging is provider-specific.

Default log level

If the default log level is not set, the default log level value is `Information`.

For example, consider the following worker service app:

- Created with the .NET Worker templates.
- `appsettings.json` and `appsettings.Development.json` deleted or renamed.

With the preceding setup, navigating to the privacy or home page produces many `Trace`, `Debug`, and `Information` messages with `Microsoft` in the category name.

The following code sets the default log level when the default log level is not set in configuration:

C#

```
HostApplicationBuilder builder = Host.CreateApplicationBuilder(args);

builder.Logging.SetMinimumLevel(LogLevel.Warning);

using IHost host = builder.Build();

await host.RunAsync();
```

Filter function

A filter function is invoked for all providers and categories that don't have rules assigned to them by configuration or code:

```
C#  
  
HostApplicationBuilder builder = Host.CreateApplicationBuilder(args);  
  
builder.Logging.AddFilter((provider, category, logLevel) =>  
{  
    return provider.Contains("ConsoleLoggerProvider")  
    && (category.Contains("Example") || category.Contains("Microsoft"))  
    && logLevel >= LogLevel.Information;  
});  
  
using IHost host = builder.Build();  
  
await host.RunAsync();
```

The preceding code displays console logs when the category contains `Example` or `Microsoft` and the log level is `Information` or higher.

Log scopes

A scope groups a set of logical operations. This grouping can be used to attach the same data to each log that's created as part of a set. For example, every log created as part of processing a transaction can include the transaction ID.

A scope:

- Is an [IDisposable](#) type that's returned by the [BeginScope](#) method.
- Lasts until it's disposed.

The following providers support scopes:

- [Console](#)
- [AzureAppServicesFile](#) and [AzureAppServicesBlob](#)

Use a scope by wrapping logger calls in a `using` block:

```
C#  
  
public async Task<T> GetAsync<T>(string id)  
{  
    T result;  
    var transactionId = Guid.NewGuid().ToString();
```

```

    using (_logger.BeginScope(new List<KeyValuePair<string, object>>
    {
        new KeyValuePair<string, object>("TransactionId",
transactionId),
    }))
{
    _logger.LogInformation(
        AppLogEvents.Read, "Reading value for {Id}", id);

    var result = await _repository.GetAsync(id);
    if (result is null)
    {
        _logger.LogWarning(
            AppLogEvents.ReadNotFound, "GetAsync({Id}) not found", id);
    }
}

return result;
}

```

The following JSON enables scopes for the console provider:

JSON

```
{
    "Logging": {
        "Debug": {
            "LogLevel": {
                "Default": "Information"
            }
        },
        "Console": {
            "IncludeScopes": true,
            "LogLevel": {
                "Microsoft": "Warning",
                "Default": "Information"
            }
        },
        "LogLevel": {
            "Default": "Debug"
        }
    }
}
```

The following code enables scopes for the console provider:

C#

```

HostApplicationBuilder builder = Host.CreateApplicationBuilder(args);

builder.Logging.ClearProviders();

```

```
builder.Logging.AddConsole(options => options.IncludeScopes = true);

using IHost host = builder.Build();

await host.RunAsync();
```

Create logs in Main

The following code logs in `Main` by getting an `ILogger` instance from DI after building the host:

C#

```
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
using Microsoft.Extensions.Logging;

using IHost host = Host.CreateApplicationBuilder(args).Build();

var logger = host.Services.GetRequiredService<ILogger<Program>>();
logger.LogInformation("Host created.");

await host.RunAsync();
```

The preceding code relies on two NuGet packages:

- [Microsoft.Extensions.Hosting](#)
- [Microsoft.Extensions.Logging](#)

Its project file would look similar to the following:

XML

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net7.0</TargetFramework>
    <ImplicitUsings>enable</ImplicitUsings>
    <Nullable>enable</Nullable>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.Extensions.Hosting" Version="7.0.1" />
    <PackageReference Include="Microsoft.Extensions.Logging" Version="7.0.0" />
  </ItemGroup>
```

```
</Project>
```

No asynchronous logger methods

Logging should be so fast that it isn't worth the performance cost of asynchronous code. If a logging datastore is slow, don't write to it directly. Consider writing the log messages to a fast store initially, then moving them to the slow store later. For example, when logging to SQL Server, don't do so directly in a `Log` method, since the `Log` methods are synchronous. Instead, synchronously add log messages to an in-memory queue and have a background worker pull the messages out of the queue to do the asynchronous work of pushing data to SQL Server.

Change log levels in a running app

The Logging API doesn't include a scenario to change log levels while an app is running. However, some configuration providers are capable of reloading configuration, which takes immediate effect on logging configuration. For example, the [File Configuration Provider](#) reloads logging configuration by default. If the configuration is changed in code while an app is running, the app can call [IConfigurationRoot.Reload](#) to update the app's logging configuration.

NuGet packages

The [ILogger<TCategoryName>](#) and [ILoggerFactory](#) interfaces and implementations are included in the .NET SDK. They are also available in the following NuGet packages:

- The interfaces are in [Microsoft.Extensions.Logging.Abstractions](#).
- The default implementations are in [Microsoft.Extensions.Logging](#).

See also

- [Logging providers in .NET](#)
- [Implement a custom logging provider in .NET](#)
- [Console log formatting](#)
- [High-performance logging in .NET](#)
- [Logging guidance for .NET library authors](#)
- Logging bugs should be created in the [github.com/dotnet/runtime](#) repo

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

.NET is an open source project.
Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

Logging providers in .NET

Article • 06/23/2023

Logging providers persist logs, except for the `Console` provider, which only displays logs as standard output. For example, the Azure Application Insights provider stores logs in Azure Application Insights. Multiple providers can be enabled.

The default .NET Worker app templates:

- Use the [Generic Host](#).
- Call [CreateApplicationBuilder](#), which adds the following logging providers:
 - [Console](#)
 - [Debug](#)
 - [EventSource](#)
 - [EventLog](#) (Windows only)

C#

```
using Microsoft.Extensions.Hosting;

using IHost host = Host.CreateApplicationBuilder(args).Build();

// Application code should start here.

await host.RunAsync();
```

The preceding code shows the `Program` class created with the .NET Worker app templates. The next several sections provide samples based on the .NET Worker app templates, which use the Generic Host.

To override the default set of logging providers added by

`Host.CreateApplicationBuilder`, call `ClearProviders` and add the logging providers you want. For example, the following code:

- Calls [ClearProviders](#) to remove all the `ILoggerProvider` instances from the builder.
- Adds the [Console](#) logging provider.

C#

```
HostApplicationBuilder builder = Host.CreateApplicationBuilder(args);

builder.Logging.ClearProviders();
builder.Logging.AddConsole();
```

For additional providers, see:

- [Built-in logging providers](#).
- [Third-party logging providers](#).

Configure a service that depends on `ILogger`

To configure a service that depends on `ILogger<T>`, use constructor injection or provide a factory method. The factory method approach is recommended only if there's no other option. For example, consider a service that needs an `ILogger<T>` instance provided by DI:

C#

```
HostApplicationBuilder builder = Host.CreateApplicationBuilder(args);

builder.Services.AddSingleton<IExampleService>(
    container => new DefaultExampleService
    {
        Logger = container.GetRequiredService<ILogger<IExampleService>>()
    });

```

The preceding code is a `Func<IServiceProvider, IExampleService>` that runs the first time the DI container needs to construct an instance of `IExampleService`. You can access any of the registered services in this way.

Built-in logging providers

Microsoft Extensions include the following logging providers as part of the runtime libraries:

- [Console](#)
- [Debug](#)
- [EventSource](#)
- [EventLog](#)

The following logging providers are shipped by Microsoft, but not as part of the runtime libraries. They must be installed as additional NuGet packages.

- [AzureAppServicesFile](#) and [AzureAppServicesBlob](#)
- [ApplicationInsights](#)

Console

The `Console` provider logs output to the console.

Debug

The `Debug` provider writes log output by using the `System.Diagnostics.Debug` class, specifically through the `Debug.WriteLine` method and only when the debugger is attached. The `DebugLoggerProvider` creates `DebugLogger` instances, which are implementations of the `ILogger` interface.

Event Source

The `EventSource` provider writes to a cross-platform event source with the name `Microsoft.Extensions.Logging`. On Windows, the provider uses `ETW`.

dotnet trace tooling

The `dotnet-trace` tool is a cross-platform CLI global tool that enables the collection of .NET Core traces of a running process. The tool collects `Microsoft.Extensions.Logging.EventSource` provider data using a `LoggingEventSource`.

See `dotnet-trace` for installation instructions. For a diagnostic tutorial using `dotnet-trace`, see [Debug high CPU usage in .NET Core](#).

Windows EventLog

The `EventLog` provider sends log output to the Windows Event Log. Unlike the other providers, the `EventLog` provider does **not** inherit the default non-provider settings. If `EventLog` log settings aren't specified, they default to `LogLevel.Warning`.

To log events lower than `LogLevel.Warning`, explicitly set the log level. The following example sets the Event Log default log level to `LogLevel.Information`:

JSON

```
"Logging": {  
    "EventLog": {  
        "LogLevel": {  
            "Default": "Information"  
        }  
    }  
}
```

`AddEventLog` overloads can pass in `EventLogSettings`. If `null` or not specified, the following default settings are used:

- `LogName`: "Application"
- `SourceName`: ".NET Runtime"
- `MachineName`: The local machine name is used.

The following code changes the `SourceName` from the default value of ".NET Runtime" to `CustomLogs`:

C#

```
HostApplicationBuilder builder = Host.CreateApplicationBuilder(args);

builder.Logging.AddEventLog(
    config => config.SourceName = "CustomLogs");

using IHost host = builder.Build();

host.Run();
```

Azure App Service

The [Microsoft.Extensions.Logging.AzureAppServices](#) provider package writes logs to text files in an Azure App Service app's file system and to [blob storage](#) in an Azure Storage account.

The provider package isn't included in the runtime libraries. To use the provider, add the provider package to the project.

To configure provider settings, use [AzureFileLoggerOptions](#) and [AzureBlobLoggerOptions](#), as shown in the following example:

C#

```
using Microsoft.Extensions.Logging.AzureAppServices;

HostApplicationBuilder builder = Host.CreateApplicationBuilder(args)

builder.Logging.AddAzureWebAppDiagnostics();
builder.Services.Configure<AzureFileLoggerOptions>(options =>
{
    options.FileName = "azure-diagnostics-";
    options.FileSizeLimit = 50 * 1024;
    options.RetainedFileCountLimit = 5;
});
builder.ServicesConfigure<AzureBlobLoggerOptions>(options =>
```

```
{  
    options.BlobName = "log.txt";  
});  
  
using IHost host = builder.Build();  
  
// Application code should start here.  
  
await host.RunAsync();
```

When deployed to Azure App Service, the app uses the settings in the [App Service logs](#) section of the [App Service](#) page of the Azure portal. When the following settings are updated, the changes take effect immediately without requiring a restart or redeployment of the app.

The default location for log files is in the *D:\home\LogFiles\Application* folder. Additional defaults vary by provider:

- **Application Logging (Filesystem):** The default filesystem file name is *diagnostics-yyyymmdd.txt*. The default file size limit is 10 MB, and the default maximum number of files retained is 2.
- **Application Logging (Blob):** The default blob name is *{app-name}/yyyy/mm/dd/hh/{guid}_applicationLog.txt*.

This provider only logs when the project runs in the Azure environment.

Azure log streaming

Azure log streaming supports viewing log activity in real-time from:

- The app server
- The web server
- Failed request tracing

To configure Azure log streaming:

- Navigate to the [App Service logs](#) page from the app's portal page.
- Set **Application Logging (Filesystem)** to **On**.
- Choose the log **Level**. This setting only applies to Azure log streaming.

Navigate to the [Log Stream](#) page to view logs. The logged messages are logged with the [ILogger](#) interface.

Azure Application Insights

The [Microsoft.Extensions.Logging.ApplicationInsights](#) provider package writes logs to [Azure Application Insights](#). Application Insights is a service that monitors a web app and provides tools for querying and analyzing the telemetry data. If you use this provider, you can query and analyze your logs by using the Application Insights tools.

For more information, see the following resources:

- [Application Insights overview](#)
- [ApplicationInsightsLoggerProvider for .NET Core ILogger logs](#) - Start here if you want to implement the logging provider without the rest of Application Insights telemetry.
- [Application Insights logging adapters](#).
- [Install, configure, and initialize the Application Insights SDK](#) - Interactive tutorial on the Microsoft Learn site.

Logging provider design considerations

If you plan to develop your own implementation of the [ILoggerProvider](#) interface and corresponding custom implementation of [ILogger](#), consider the following points:

- The [ILogger.Log](#) method is synchronous.
- The lifetime of log state and objects should *not* be assumed.

An implementation of [ILoggerProvider](#) will create an [ILogger](#) via its [ILoggerProvider.CreateLogger](#) method. If your implementation strives to queue logging messages in a non-blocking manner, the messages should first be materialized or the object state that's used to materialize a log entry should be serialized. Doing so avoids potential exceptions from disposed objects.

For more information, see [Implement a custom logging provider in .NET](#).

Third-party logging providers

Here are some third-party logging frameworks that work with various .NET workloads:

- [elmah.io](#) ([GitHub repo](#))
- [Gelf](#) ([GitHub repo](#))
- [JSNLog](#) ([GitHub repo](#))
- [KissLog.net](#) ([GitHub repo](#))
- [Log4Net](#) ([GitHub repo](#))
- [NLog](#) ([GitHub repo](#))
- [NReco.Logging](#) ([GitHub repo](#))

- [Sentry](#) ([GitHub repo](#))
- [Serilog](#) ([GitHub repo](#))
- [Stackdriver](#) ([GitHub repo](#))

Some third-party frameworks can perform [semantic logging, also known as structured logging](#).

Using a third-party framework is similar to using one of the built-in providers:

1. Add a NuGet package to your project.
2. Call an `ILoggerFactory` or `ILoggingBuilder` extension method provided by the logging framework.

For more information, see each provider's documentation. Third-party logging providers aren't supported by Microsoft.

See also

- [Logging in .NET](#).
- [Implement a custom logging provider in .NET](#).
- [High-performance logging in .NET](#).

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

Compile-time logging source generation

Article • 10/11/2023

.NET 6 introduces the `LoggerMessageAttribute` type. This attribute is part of the `Microsoft.Extensions.Logging` namespace, and when used, it source-generates performant logging APIs. The source-generation logging support is designed to deliver a highly usable and highly performant logging solution for modern .NET applications. The auto-generated source code relies on the `ILogger` interface in conjunction with `LoggerMessage.Define` functionality.

The source generator is triggered when `LoggerMessageAttribute` is used on `partial` logging methods. When triggered, it is either able to autogenerate the implementation of the `partial` methods it's decorating, or produce compile-time diagnostics with hints about proper usage. The compile-time logging solution is typically considerably faster at run time than existing logging approaches. It achieves this by eliminating boxing, temporary allocations, and copies to the maximum extent possible.

Basic usage

To use the `LoggerMessageAttribute`, the consuming class and method need to be `partial`. The code generator is triggered at compile time, and generates an implementation of the `partial` method.

C#

```
public static partial class Log
{
    [LoggerMessage(
        EventId = 0,
        Level = LogLevel.Critical,
        Message = "Could not open socket to `{HostName}`")]
    public static partial void CouldNotOpenSocket(
        ILogger logger, string hostName);
}
```

In the preceding example, the logging method is `static` and the log level is specified in the attribute definition. When using the attribute in a static context, either the `ILogger` instance is required as a parameter, or modify the definition to use the `this` keyword to define the method as an extension method.

C#

```
public static partial class Log
{
    [LoggerMessage(
        EventId = 0,
        Level = LogLevel.Critical,
        Message = "Could not open socket to `{HostName}`")]
    public static partial void CouldNotOpenSocket(
        this ILogger logger, string hostName);
}
```

You may choose to use the attribute in a non-static context as well. Consider the following example where the logging method is declared as an instance method. In this context, the logging method gets the logger by accessing an `ILogger` field in the containing class.

C#

```
public partial class InstanceLoggingExample
{
    private readonly ILogger _logger;

    public InstanceLoggingExample(ILogger logger)
    {
        _logger = logger;
    }

    [LoggerMessage(
        EventId = 0,
        Level = LogLevel.Critical,
        Message = "Could not open socket to `{HostName}`")]
    public partial void CouldNotOpenSocket(string hostName);
}
```

Sometimes, the log level needs to be dynamic rather than statically built into the code. You can do this by omitting the log level from the attribute and instead requiring it as a parameter to the logging method.

C#

```
public static partial class Log
{
    [LoggerMessage(
        EventId = 0,
        Message = "Could not open socket to `{HostName}`")]
    public static partial void CouldNotOpenSocket(
        ILogger logger,
        LogLevel level, /* Dynamic log level as parameter, rather than
```

```
defined in attribute. */
        string hostName);
}
```

You can omit the logging message and `String.Empty` will be provided for the message. The state will contain the arguments, formatted as key-value pairs.

C#

```
using System.Text.Json;
using Microsoft.Extensions.Logging;

using ILoggerFactory loggerFactory = LoggerFactory.Create(
    builder =>
    builder.AddJsonConsole(
        options =>
        options.JsonWriterOptions = new JsonWriterOptions()
        {
            Indented = true
        }));
    });

ILogger<SampleObject> logger = loggerFactory.CreateLogger<SampleObject>();
logger.PlaceOfResidence(logLevel: LogLevel.Information, name: "Liana", city:
    "Seattle");

readonly file record struct SampleObject { }

public static partial class Log
{
    [LoggerMessage(EventId = 23, Message = "{Name} lives in {City}.")]
    public static partial void PlaceOfResidence(
        this ILogger logger,
        LogLevel logLevel,
        string name,
        string city);
}
```

Consider the example logging output when using the `JsonConsole` formatter.

JSON

```
{
    "EventId": 23,
    "LogLevel": "Information",
    "Category": "\u003CProgram\u003EF...9CB42__SampleObject",
    "Message": "Liana lives in Seattle.",
    "State": {
        "Message": "Liana lives in Seattle.",
        "name": "Liana",
        "city": "Seattle",
        "{OriginalFormat)": "{Name} lives in {City}."}
```

```
}
```

Log method constraints

When using the `LoggerMessageAttribute` on logging methods, some constraints must be followed:

- Logging methods must be `partial` and return `void`.
- Logging method names must *not* start with an underscore.
- Parameter names of logging methods must *not* start with an underscore.
- Logging methods may *not* be defined in a nested type.
- Logging methods *cannot* be generic.
- If a logging method is `static`, the `ILogger` instance is required as a parameter.

The code-generation model depends on code being compiled with a modern C# compiler, version 9 or later. The C# 9.0 compiler became available with .NET 5. To upgrade to a modern C# compiler, edit your project file to target C# 9.0.

XML

```
<PropertyGroup>
  <LangVersion>9.0</LangVersion>
</PropertyGroup>
```

For more information, see [C# language versioning](#).

Log method anatomy

The `ILogger.Log` signature accepts the `LogLevel` and optionally an `Exception`, as shown below.

C#

```
public interface ILogger
{
    void Log<TState>(
        Microsoft.Extensions.Logging.LogLevel logLevel,
        Microsoft.Extensions.Logging.EventId eventId,
        TState state,
        System.Exception? exception,
        Func<TState, System.Exception?, string> formatter);
}
```

As a general rule, the first instance of `ILogger`, `LogLevel`, and `Exception` are treated specially in the log method signature of the source generator. Subsequent instances are treated like normal parameters to the message template:

C#

```
// This is a valid attribute usage
[LoggerMessage(
    EventId = 110, Level = LogLevel.Debug, Message = "M1 {Ex3} {Ex2}")]
public static partial void ValidLogMethod(
    ILogger logger,
    Exception ex,
    Exception ex2,
    Exception ex3);

// This causes a warning
[LoggerMessage(
    EventId = 0, Level = LogLevel.Debug, Message = "M1 {Ex} {Ex2}")]
public static partial void WarningLogMethod(
    ILogger logger,
    Exception ex,
    Exception ex2);
```

ⓘ Important

The warnings emitted provide details as to the correct usage of the `LoggerMessageAttribute`. In the preceding example, the `WarningLogMethod` will report a `DiagnosticSeverity.Warning` of `SYSLIB0025`.

Console

Don't include a template for `ex` in the logging message since it is implicitly taken care of.

Case-insensitive template name support

The generator does a case-insensitive comparison between items in the message template and argument names in the log message. This means that when the `ILogger` enumerates the state, the argument is picked up by the message template, which can make the logs nicer to consume:

C#

```
public partial class LoggingExample
{
```

```

private readonly ILogger _logger;

public LoggingExample(ILogger logger)
{
    _logger = logger;
}

[LoggerMessage(
    EventId = 10,
    Level = LogLevel.Information,
    Message = "Welcome to {City} {Province}!")]
public partial void LogMethodSupportsPascalCasingOfNames(
    string city, string province);

public void TestLogging()
{
    LogMethodSupportsPascalCasingOfNames("Vancouver", "BC");
}
}

```

Consider the example logging output when using the `JsonConsole` formatter:

JSON

```
{
    "EventId": 13,
    "LogLevel": "Information",
    "Category": "LoggingExample",
    "Message": "Welcome to Vancouver BC!",
    "State": {
        "Message": "Welcome to Vancouver BC!",
        "City": "Vancouver",
        "Province": "BC",
        "{OriginalFormat)": "Welcome to {City} {Province}!"
    }
}
```

Indeterminate parameter order

There are no constraints on the ordering of log method parameters. A developer could define the `ILogger` as the last parameter, although it may appear a bit awkward.

C#

```

[LoggerMessage(
    EventId = 110,
    Level = LogLevel.Debug,
    Message = "M1 {Ex3} {Ex2}")]
static partial void LogMethod(
    Exception ex,

```

```
Exception ex2,  
Exception ex3,  
ILogger logger);
```

💡 Tip

The order of the parameters on a log method is *not* required to correspond to the order of the template placeholders. Instead, the placeholder names in the template are expected to match the parameters. Consider the following `JsonConsole` output and the order of the errors.

JSON

```
{  
    "EventId": 110,  
    "LogLevel": "Debug",  
    "Category": "ConsoleApp.Program",  
    "Message": "M1 System.Exception: Third time's the charm.  
System.Exception: This is the second error.",  
    "State": {  
        "Message": "M1 System.Exception: Third time's the charm.  
System.Exception: This is the second error.",  
        "ex2": "System.Exception: This is the second error.",  
        "ex3": "System.Exception: Third time's the charm.",  
        "{OriginalFormat)": "M1 {Ex3} {Ex2}"  
    }  
}
```

Additional logging examples

The following samples demonstrate how to retrieve the event name, set the log level dynamically, and format logging parameters. The logging methods are:

- `LogWithCustomEventName`: Retrieve event name via `LoggerMessage` attribute.
- `LogWithDynamicLogLevel`: Set log level dynamically, to allow log level to be set based on configuration input.
- `UsingFormatSpecifier`: Use format specifiers to format logging parameters.

C#

```
public partial class LoggingSample  
{  
    private readonly ILogger _logger;  
  
    public LoggingSample(ILogger logger)
```

```

    {
        _logger = logger;
    }

    [LoggerMessage(
        EventId = 20,
        Level = LogLevel.Critical,
        Message = "Value is {Value:E}")]
    public static partial void UsingFormatSpecifier(
        ILogger logger, double value);

    [LoggerMessage(
        EventId = 9,
        Level = LogLevel.Trace,
        Message = "Fixed message",
        EventName = "CustomEventName")]
    public partial void LogWithCustomEventName();

    [LoggerMessage(
        EventId = 10,
        Message = "Welcome to {City} {Province}!")]
    public partial void LogWithDynamicLogLevel(
        string city, LogLevel level, string province);

    public void TestLogging()
    {
        LogWithCustomEventName();

        LogWithDynamicLogLevel("Vancouver", LogLevel.Warning, "BC");
        LogWithDynamicLogLevel("Vancouver", LogLevel.Information, "BC");

        UsingFormatSpecifier(logger, 12345.6789);
    }
}

```

Consider the example logging output when using the `SimpleConsole` formatter:

Console

```

trce: LoggingExample[9]
    Fixed message
warn: LoggingExample[10]
    Welcome to Vancouver BC!
info: LoggingExample[10]
    Welcome to Vancouver BC!
crit: LoggingExample[20]
    Value is 1.234568E+004

```

Consider the example logging output when using the `JsonConsole` formatter:

JSON

```
{
    "EventId": 9,
    "LogLevel": "Trace",
    "Category": "LoggingExample",
    "Message": "Fixed message",
    "State": {
        "Message": "Fixed message",
        "{OriginalFormat}": "Fixed message"
    }
}
{
    "EventId": 10,
    "LogLevel": "Warning",
    "Category": "LoggingExample",
    "Message": "Welcome to Vancouver BC!",
    "State": {
        "Message": "Welcome to Vancouver BC!",
        "city": "Vancouver",
        "province": "BC",
        "{OriginalFormat}": "Welcome to {City} {Province}!"
    }
}
{
    "EventId": 10,
    "LogLevel": "Information",
    "Category": "LoggingExample",
    "Message": "Welcome to Vancouver BC!",
    "State": {
        "Message": "Welcome to Vancouver BC!",
        "city": "Vancouver",
        "province": "BC",
        "{OriginalFormat}": "Welcome to {City} {Province}!"
    }
}
{
    "EventId": 20,
    "LogLevel": "Critical",
    "Category": "LoggingExample",
    "Message": "Value is 1.234568E+004",
    "State": {
        "Message": "Value is 1.234568E+004",
        "value": 12345.6789,
        "{OriginalFormat}": "Value is {Value:E}"
    }
}
}
```

Summary

With the advent of C# source generators, writing highly performant logging APIs is much easier. Using the source generator approach has several key benefits:

- Allows the logging structure to be preserved and enables the exact format syntax required by [Message Templates](#).
- Allows supplying alternative names for the template placeholders and using format specifiers.
- Allows the passing of all original data as-is, without any complication around how it's stored before something is done with it (other than creating a `string`).
- Provides logging-specific diagnostics, and emits warnings for duplicate event IDs.

Additionally, there are benefits over manually using `LoggerMessage.Define`:

- Shorter and simpler syntax: Declarative attribute usage rather than coding boilerplate.
- Guided developer experience: The generator gives warnings to help developers do the right thing.
- Support for an arbitrary number of logging parameters. `LoggerMessage.Define` supports a maximum of six.
- Support for dynamic log level. This is not possible with `LoggerMessage.Define` alone.

See also

- [Logging in .NET](#)
- [High-performance logging in .NET](#)
- [Console log formatting](#)
- [NuGet: Microsoft.Extensions.Logging.Abstractions](#)

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

Logging guidance for .NET library authors

Article • 10/11/2023

As a library author, exposing logging is a great way to provide consumers with insight into the inner workings of your library. This guidance helps you expose logging in a way that is consistent with other .NET libraries and frameworks. It also helps you avoid common performance bottlenecks that may not be otherwise obvious.

When to use the `ILoggerFactory` interface

When writing a library that emits logs, you need an `ILogger` object to record the logs. To get that object, your API can either accept an `ILogger<TCategoriesName>` parameter, or it can accept an `ILoggerFactory` after which you call `ILoggerFactory.CreateLogger`. Which approach should be preferred?

- When you need a logging object that can be passed along to multiple classes so that all of them can emit logs, use `ILoggerFactory`. It's recommended that each class creates logs with a separate category, named the same as the class. To do this, you need the factory to create unique `ILogger<TCategoriesName>` objects for each class that emits logs. Common examples include public entry point APIs for a library or public constructors of types that might create helper classes internally.
- When you need a logging object that's only used inside one class and never shared, use `ILogger<TCategoriesName>`, where `TCategoriesName` is the type that produces the logs. A common example of this is a constructor for a class created by dependency injection.

If you're designing a public API that must remain stable over time, keep in mind that you might desire to refactor your internal implementation in the future. Even if a class doesn't create any internal helper types initially, that might change as the code evolves. Using `ILoggerFactory` accommodates creating new `ILogger<TCategoriesName>` objects for any new classes without changing the public API.

For more information, see [How filtering rules are applied](#).

Prefer source-generated logging

The `ILogger` API supports two approaches to using the API. You can either call methods such as `LoggerExtensions.LogError` and `LoggerExtensions.LogInformation`, or you can use the logging source generator to define strongly typed logging methods. For most situations, the source generator is recommended because it offers superior performance and stronger typing. It also isolates logging-specific concerns such as message templates, IDs, and log levels from the calling code. The non-source-generated approach is primarily useful for scenarios where you are willing to give up those advantages to make the code more concise.

C#

```
using Microsoft.Extensions.Logging;

namespace Logging.LibraryAuthors;

internal static partial class LogMessages
{
    [LoggerMessage(
        Message = "Sold {Quantity} of {Description}",
        Level = LogLevel.Information)]
    internal static partial void LogProductSaleDetails(
        this ILogger logger,
        int quantity,
        string description);
}
```

The preceding code:

- Defines a `partial class` named `LogMessages`, which is `static` so that it can be used to define extension methods on the `ILogger` type.
- Decorates a `LogProductSaleDetails` extension method with the `LoggerMessage` attribute and `Message` template.
- Declares `LogProductSaleDetails`, which extends the `ILogger` and accepts a `quantity` and `description`.

💡 Tip

You can step into the source-generated code during debugging, because it's part of the same assembly as the code that calls it.

Use `.IsEnabled` to avoid expensive parameter evaluation

There may be situations where evaluating parameters is expensive. Expanding upon the previous example, imagine the `description` parameter is a `string` that is expensive to compute. Perhaps the product being sold gets a friendly product description and relies on a database query, or reading from a file. In these situations, you can instruct the source generator to skip the `.IsEnabled` guard and manually add the `.IsEnabled` guard at the call site. This allows the user to determine where the guard is called and ensures that parameters that might be expensive to compute are only evaluated when truly needed. Consider the following code:

```
C#  
  
using Microsoft.Extensions.Logging;  
  
namespace Logging.LibraryAuthors;  
  
internal static partial class LogMessages  
{  
    [LoggerMessage(  
        Message = "Sold {Quantity} of {Description}",  
        Level = LogLevel.Information,  
        SkipEnabledCheck = true)]  
    internal static partial void LogProductSaleDetails(  
        this ILogger logger,  
        int quantity,  
        string description);  
}
```

When the `LogProductSaleDetails` extension method is called, the `.IsEnabled` guard is manually invoked and the expensive parameter evaluation is limited to when it's needed. Consider the following code:

```
C#  
  
if (_logger.IsEnabled(LogLevel.Information))  
{  
    // Expensive parameter evaluation  
    var description = product.GetFriendlyProductDescription();  
  
    _logger.LogProductSaleDetails(  
        quantity,  
        description);  
}
```

For more information, see [Compile-time logging source generation](#) and [High-performance logging in .NET](#).

Avoid string interpolation in logging

A common mistake is to use [string interpolation](#) to build log messages. String interpolation in logging is problematic for performance, as the string is evaluated even if the corresponding `LogLevel` isn't enabled. Instead of string interpolation, use the log message template, formatting, and argument list. For more information, see [Logging in .NET: Log message template](#).

Use no-op logging defaults

Libraries can default to *null logging* if no `ILoggerFactory` is provided. The use of *null logging* differs from defining types as nullable (`ILoggerFactory?`), as the types are non-null. These convenience-based types don't log anything and are essentially no-ops. Consider using any of the available abstraction types where applicable:

- [NullLogger.Instance](#)
- [NullLoggerFactory.Instance](#)
- [NullLoggerProvider.Instance](#)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback [here](#).

 [Open a documentation issue](#)

 [Provide product feedback](#)

Implement a custom logging provider in .NET

Article • 03/18/2023

There are many [logging providers](#) available for common logging needs. You may need to implement a custom [ILoggerProvider](#) when one of the available providers doesn't suit your application needs. In this article, you'll learn how to implement a custom logging provider that can be used to colorize logs in the console.

💡 Tip

The custom logging provider example source code is available in the [Docs Github repo](#). For more information, see [GitHub: .NET Docs - Console Custom Logging](#).

Sample custom logger configuration

The sample creates different color console entries per log level and event ID using the following configuration type:

C#

```
using Microsoft.Extensions.Logging;

public sealed class ColorConsoleLoggerConfiguration
{
    public int EventId { get; set; }

    public Dictionary<LogLevel, ConsoleColor> LogLevelToColorMap { get; set; }
} = new()
{
    [LogLevel.Information] = ConsoleColor.Green
};
```

The preceding code sets the default level to `Information`, the color to `Green`, and the `EventId` is implicitly `0`.

Create the custom logger

The `ILogger` implementation category name is typically the logging source. For example, the type where the logger is created:

C#

```
using Microsoft.Extensions.Logging;

public sealed class ColorConsoleLogger : ILogger
{
    private readonly string _name;
    private readonly Func<ColorConsoleLoggerConfiguration>
    _getCurrentConfig;

    public ColorConsoleLogger(
        string name,
        Func<ColorConsoleLoggerConfiguration> getCurrentConfig) =>
        (_name, _getCurrentConfig) = (name, getCurrentConfig);

    public IDisposable? BeginScope<TState>(TState state) where TState : notnull => default!;

    public bool IsEnabled(LogLevel logLevel) =>
        _getCurrentConfig().LogLevelToColorMap.ContainsKey(logLevel);

    public void Log<TState>(
        LogLevel logLevel,
        EventId eventId,
        TState state,
        Exception? exception,
        Func<TState, Exception?, string> formatter)
    {
        if (!IsEnabled(logLevel))
        {
            return;
        }

        ColorConsoleLoggerConfiguration config = _getCurrentConfig();
        if (config.EventId == 0 || config.EventId == eventId.Id)
        {
            ConsoleColor originalColor = Console.ForegroundColor;

            Console.ForegroundColor = config.LogLevelToColorMap[logLevel];
            Console.WriteLine($"[{eventId.Id,2}: {logLevel,-12}]");

            Console.ForegroundColor = originalColor;
            Console.Write($"{_name} - ");
        }

        Console.ForegroundColor = config.LogLevelToColorMap[logLevel];
        Console.Write($"{formatter(state, exception)}");

        Console.ForegroundColor = originalColor;
        Console.WriteLine();
    }
}
```

The preceding code:

- Creates a logger instance per category name.
- Checks `_getCurrentConfig().LogLevelToColorMap.ContainsKey(logLevel)` in `IsEnabled`, so each `logLevel` has a unique logger. In this implementation, each log level requires an explicit configuration entry to log.

It's a good practice to call `ILogger.IsEnabled` within `ILogger.Log` implementations since `Log` can be called by any consumer, and there are no guarantees that it was previously checked. The `IsEnabled` method should be very fast in most implementations.

C#

```
public bool IsEnabled(LogLevel logLevel) =>
    _getCurrentConfig().LogLevelToColorMap.ContainsKey(logLevel);
```

The logger is instantiated with the `name` and a `Func<ColorConsoleLoggerConfiguration>`, which returns the current config — this handles updates to the config values as monitored through the `IOptionsMonitor<TOptions>.OnChange` callback.

ⓘ Important

The `ILogger.Log` implementation checks if the `config.EventId` value is set. When `config.EventId` is not set or when it matches the exact `logEntry.EventId`, the logger logs in color.

Custom logger provider

The `ILoggerProvider` object is responsible for creating logger instances. It's not necessary to create a logger instance per category, but it makes sense for some loggers, like NLog or log4net. This strategy allows you to choose different logging output targets per category, as in the following example:

C#

```
using System.Collections.Concurrent;
using System.Runtime.Versioning;
using Microsoft.Extensions.Logging;
using Microsoft.Extensions.Options;

[UnsupportedOSPlatform("browser")]
[ProviderAlias("ColorConsole")]
public sealed class ColorConsoleLoggerProvider : ILoggerProvider
```

```

{
    private readonly IDisposable? _onChangeToken;
    private ColorConsoleLoggerConfiguration _currentConfig;
    private readonly ConcurrentDictionary<string, ColorConsoleLogger>
    _loggers =
        new(StringComparer.OrdinalIgnoreCase);

    public ColorConsoleLoggerProvider(
        IOptionsMonitor<ColorConsoleLoggerConfiguration> config)
    {
        _currentConfig = config.CurrentValue;
        _onChangeToken = config.OnChange(updatedConfig => _currentConfig =
updatedConfig);
    }

    public ILogger CreateLogger(string categoryName) =>
        _loggers.GetOrAdd(categoryName, name => new ColorConsoleLogger(name,
GetCurrentConfig()));

    private ColorConsoleLoggerConfiguration GetCurrentConfig() =>
        _currentConfig;

    public void Dispose()
    {
        _loggers.Clear();
        _onChangeToken?.Dispose();
    }
}

```

In the preceding code, `CreateLogger` creates a single instance of the `ColorConsoleLogger` per category name and stores it in the `ConcurrentDictionary< TKey, TValue >`. Additionally, the `IOptionsMonitor< TOptions >` interface is required to update changes to the underlying `ColorConsoleLoggerConfiguration` object.

To control the configuration of the `ColorConsoleLogger`, you define an alias on its provider:

C#

```

[UnsupportedOSPlatform("browser")]
[ProviderAlias("ColorConsole")]
public sealed class ColorConsoleLoggerProvider : ILoggerProvider

```

The `ColorConsoleLoggerProvider` class defines two class-scoped attributes:

- **UnsupportedOSPlatformAttribute**: The `ColorConsoleLogger` type is *not supported* in the "browser".

- **ProviderAliasAttribute**: Configuration sections can define options using the `"ColorConsole"` key.

The configuration can be specified with any valid [configuration provider](#). Consider the following `appsettings.json` file:

JSON

```
{  
  "Logging": {  
    "ColorConsole": {  
      "LogLevelToColorMap": {  
        "Information": "DarkGreen",  
        "Warning": "Cyan",  
        "Error": "Red"  
      }  
    }  
  }  
}
```

This configures the log levels to the following values:

- `LogLevel.Information`: `ConsoleColor.DarkGreen`
- `LogLevel.Warning`: `ConsoleColor.Cyan`
- `LogLevel.Error`: `ConsoleColor.Red`

The `Information` log level is set to `DarkGreen`, which overrides the default value set in the `ColorConsoleLoggerConfiguration` object.

Usage and registration of the custom logger

By convention, registering services for dependency injection happens as part of the startup routine of an application. The registration occurs in the `Program` class, or could be delegated to a `Startup` class. In this example, you'll register directly from the `Program.cs`.

To add the custom logging provider and corresponding logger, add an [`ILoggerProvider`](#) with [`ILoggingBuilder`](#) from the `HostingHostBuilderExtensions.ConfigureLogging(IHostBuilder, Action<ILoggingBuilder>)`:

C#

```
using Microsoft.Extensions.DependencyInjection;  
using Microsoft.Extensions.Hosting;  
using Microsoft.Extensions.Logging;
```

```

HostApplicationBuilder builder = Host.CreateApplicationBuilder(args);

builder.Logging.ClearProviders();
builder.Logging.AddColorConsoleLogger(configuration =>
{
    // Replace warning value from appsettings.json of "Cyan"
    configuration.LogLevelToColorMap[LogLevel.Warning] =
    ConsoleColor.DarkCyan;
    // Replace warning value from appsettings.json of "Red"
    configuration.LogLevelToColorMap[LogLevel.Error] = ConsoleColor.DarkRed;
});

using IHost host = builder.Build();

var logger = host.Services.GetRequiredService<ILogger<Program>>();

logger.LogDebug(1, "Does this line get hit?");      // Not logged
logger.LogInformation(3, "Nothing to see here."); // Logs in
ConsoleColor.DarkGreen
logger.LogWarning(5, "Warning... that was odd."); // Logs in
ConsoleColor.DarkCyan
logger.LogError(7, "Oops, there was an error."); // Logs in
ConsoleColor.DarkRed
logger.LogTrace(5, "== 120.");                      // Not logged

await host.RunAsync();

```

The `ILoggingBuilder` creates one or more `ILogger` instances. The `ILogger` instances are used by the framework to log the information.

The configuration from the `appsettings.json` file overrides the following values:

- `LogLevel.Warning`: `ConsoleColor.DarkCyan`
- `LogLevel.Error`: `ConsoleColor.DarkRed`

By convention, extension methods on `ILoggingBuilder` are used to register the custom provider:

C#

```

using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.DependencyInjection.Extensions;
using Microsoft.Extensions.Logging;
using Microsoft.Extensions.Logging.Configuration;

public static class ColorConsoleLoggerExtensions
{
    public static ILoggingBuilder AddColorConsoleLogger(
        this ILoggingBuilder builder)
    {

```

```

        builder.AddConfiguration();

        builder.Services.TryAddEnumerable(
            ServiceDescriptor.Singleton<ILoggerProvider,
ColorConsoleLoggerProvider>());

        LoggerProviderOptions.RegisterProviderOptions
            <ColorConsoleLoggerConfiguration, ColorConsoleLoggerProvider>
(builder.Services);

        return builder;
    }

    public static ILoggingBuilder AddColorConsoleLogger(
        this ILoggingBuilder builder,
        Action<ColorConsoleLoggerConfiguration> configure)
    {
        builder.AddColorConsoleLogger();
        builder.Services.Configure(configure);

        return builder;
    }
}

```

Running this simple application will render color output to the console window similar to the following image:



See also

- [Logging in .NET](#)
- [Logging providers in .NET](#)
- [Dependency injection in .NET](#)
- [High-performance logging in .NET](#)

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

High-performance logging in .NET

Article • 03/14/2023

The [LoggerMessage](#) class exposes functionality to create cacheable delegates that require fewer object allocations and reduced computational overhead compared to [logger extension methods](#), such as [LogInformation](#) and [LogDebug](#). For high-performance logging scenarios, use the [LoggerMessage](#) pattern.

[LoggerMessage](#) provides the following performance advantages over Logger extension methods:

- Logger extension methods require "boxing" (converting) value types, such as `int`, into `object`. The [LoggerMessage](#) pattern avoids boxing by using static `Action` fields and extension methods with strongly typed parameters.
- Logger extension methods must parse the message template (named format string) every time a log message is written. [LoggerMessage](#) only requires parsing a template once when the message is defined.

ⓘ Important

Instead of using the [LoggerMessage class](#) to create high-performance logs, you can use the [LoggerMessage attribute](#) in .NET 6.0. The `LoggerMessageAttribute` provides source-generation logging support designed to deliver a highly usable and highly performant logging solution for modern .NET applications. For more information, see [Compile-time logging source generation \(.NET Fundamentals\)](#).

The sample app demonstrates [LoggerMessage](#) features with a priority queue processing worker service. The app processes work items in priority order. As these operations occur, log messages are generated using the [LoggerMessage](#) pattern.

ⓘ Tip

All of the logging example source code is available in the [Samples Browser](#) for download. For more information, see [Browse code samples: Logging in .NET](#).

Define a logger message

Use [Define\(LogLevel, EventId, String\)](#) to create an `Action` delegate for logging a message. [Define](#) overloads permit passing up to six type parameters to a named format

string (template).

The string provided to the [Define](#) method is a template and not an interpolated string. Placeholders are filled in the order that the types are specified. Placeholder names in the template should be descriptive and consistent across templates. They serve as property names within structured log data. We recommend [Pascal casing](#) for placeholder names. For example, `{Item}`, `{DateTime}`.

Each log message is an [Action](#) held in a static field created by [LoggerMessage.Define](#). For example, the sample app creates a field to describe a log message for the processing of work items:

```
C#
```

```
private static readonly Action<ILogger, Exception>
s_failedToProcessWorkItem;
```

For the [Action](#), specify:

- The log level.
- A unique event identifier ([EventId](#)) with the name of the static extension method.
- The message template (named format string).

As work items are dequeued for processing, the worker service app sets the:

- Log level to [LogLevel.Critical](#).
- Event ID to `13` with the name of the `FailedToProcessWorkItem` method.
- Message template (named format string) to a string.

```
C#
```

```
s_failedToProcessWorkItem = LoggerMessage.Define(
    LogLevel.Critical,
    new EventId(13, nameof(FailedToProcessWorkItem)),
    "Epic failure processing item!");
```

The [LoggerMessage.Define](#) method is used to configure and define an [Action](#) delegate, which represents a log message.

Structured logging stores may use the event name when it's supplied with the event ID to enrich logging. For example, [Serilog](#) uses the event name.

The [Action](#) is invoked through a strongly typed extension method. The `PriorityItemProcessed` method logs a message every time a work item is processed. `FailedToProcessWorkItem` is called if and when an exception occurs:

C#

```
protected override async Task ExecuteAsync(
    CancellationToken stoppingToken)
{
    using IDisposable? scope = _logger.ProcessingWorkScope(DateTime.Now);
    while (!stoppingToken.IsCancellationRequested)
    {
        WorkItem? nextItem = _priorityQueue.ProcessNextHighestPriority();
        try
        {
            if (nextItem is not null)
            {
                _logger.PriorityItemProcessed(nextItem);
            }
        }
        catch (Exception ex)
        {
            _loggerFailedToProcessWorkItem(ex);
        }

        await Task.Delay(1_000, stoppingToken);
    }
}
```

Inspect the app's console output:

Console

```
crit: WorkerServiceOptions.Example.Worker[13]
  Epic failure processing item!
  System.Exception: Failed to verify communications.
    at
WorkerServiceOptions.Example.Worker.ExecuteAsync(CancellationToken
stoppingToken) in
  ..\Worker.cs:line 27
```

To pass parameters to a log message, define up to six types when creating the static field. The sample app logs the work item details when processing items by defining a `WorkItem` type for the `Action` field:

C#

```
private static readonly Action<ILogger, WorkItem, Exception>
s_processingPriorityItem;
```

The delegate's log message template receives its placeholder values from the types provided. The sample app defines a delegate for adding a work item where the item

parameter is a `WorkItem`:

C#

```
s_processingPriorityItem = LoggerMessage.Define<WorkItem>(
    LogLevel.Information,
    new EventId(1, nameof(PriorityItemProcessed)),
    "Processing priority item: {Item}");
```

The static extension method for logging that a work item is being processed, `PriorityItemProcessed`, receives the work item argument value and passes it to the `Action` delegate:

C#

```
public static void PriorityItemProcessed(
    this ILogger logger, WorkItem workItem) =>
    s_processingPriorityItem(logger, workItem, default!);
```

In the worker service's `ExecuteAsync` method, `PriorityItemProcessed` is called to log the message:

C#

```
protected override async Task ExecuteAsync(
    CancellationToken stoppingToken)
{
    using IDisposable? scope = _logger.ProcessingWorkScope(DateTime.Now);
    while (!stoppingToken.IsCancellationRequested)
    {
        WorkItem? nextItem = _priorityQueue.ProcessNextHighestPriority();
        try
        {
            if (nextItem is not null)
            {
                _logger.PriorityItemProcessed(nextItem);
            }
        }
        catch (Exception ex)
        {
            _logger FailedToProcessWorkItem(ex);
        }

        await Task.Delay(1_000, stoppingToken);
    }
}
```

Inspect the app's console output:

Console

```
info: WorkerServiceOptions.Example.Worker[1]
    Processing priority item: Priority-Extreme (50db062a-9732-4418-936d-
110549ad79e4): 'Verify communications'
```

Define logger message scope

The [DefineScope\(string\)](#) method creates a `Func<TResult>` delegate for defining a [log scope](#). `DefineScope` overloads permit passing up to three type parameters to a named format string (template).

As is the case with the [Define](#) method, the string provided to the [DefineScope](#) method is a template and not an interpolated string. Placeholders are filled in the order that the types are specified. Placeholder names in the template should be descriptive and consistent across templates. They serve as property names within structured log data. We recommend [Pascal casing](#) for placeholder names. For example, `{Item}`, `{DateTime}`.

Define a [log scope](#) to apply to a series of log messages using the [DefineScope](#) method. Enable `IncludeScopes` in the console logger section of `appsettings.json`:

JSON

```
{
  "Logging": {
    "Console": {
      "IncludeScopes": true
    },
    "LogLevel": {
      "Default": "Information",
      "Microsoft": "Warning",
      "Microsoft.Hosting.Lifetime": "Information"
    }
  }
}
```

To create a log scope, add a field to hold a `Func<TResult>` delegate for the scope. The sample app creates a field called `_processingWorkScope` (`Internal/LoggerExtensions.cs`):

C#

```
private static Func<ILogger, DateTime, IDisposable?> s_processingWorkScope;
```

Use `DefineScope` to create the delegate. Up to three types can be specified for use as template arguments when the delegate is invoked. The sample app uses a message template that includes the date time in which processing started:

```
C#
```

```
s_processingWorkScope =
    LoggerMessage.DefineScope<DateTime>(
        "Processing work, started at: {DateTime}");
```

Provide a static extension method for the log message. Include any type parameters for named properties that appear in the message template. The sample app takes in a `DateTime` for a custom time stamp to log and returns `_processingWorkScope`:

```
C#
```

```
public static IDisposable? ProcessingWorkScope(
    this ILogger logger, DateTime time) =>
    s_processingWorkScope(logger, time);
```

The scope wraps the logging extension calls in a `using` block:

```
C#
```

```
protected override async Task ExecuteAsync(
    CancellationToken stoppingToken)
{
    using IDisposable? scope =
        _logger.ProcessingWorkScope(DateTime.Now);
    while (!stoppingToken.IsCancellationRequested)
    {
        WorkItem? nextItem =
            _priorityQueue.ProcessNextHighestPriority();
        try
        {
            if (nextItem is not null)
            {
                _logger.PriorityItemProcessed(nextItem);
            }
        }
        catch (Exception ex)
        {
            _loggerFailedToProcessWorkItem(ex);
        }

        await Task.Delay(1_000, stoppingToken);
    }
}
```

```
    }  
}
```

Inspect the log messages in the app's console output. The following result shows priority ordering of log messages with the log scope message included:

```
Console  
  
info: WorkerServiceOptions.Example.Worker[1]  
      => Processing work, started at: 09/25/2020 14:30:45  
      Processing priority item: Priority-Extreme (f5090ede-a337-4041-b914-f6bc0db5ae64): 'Verify communications'  
info: Microsoft.Hosting.Lifetime[0]  
      Application started. Press Ctrl+C to shut down.  
info: Microsoft.Hosting.Lifetime[0]  
      Hosting environment: Development  
info: Microsoft.Hosting.Lifetime[0]  
      Content root path: ..\worker-service-options  
info: WorkerServiceOptions.Example.Worker[1]  
      => Processing work, started at: 09/25/2020 14:30:45  
      Processing priority item: Priority-High (496d440f-2007-4391-b179-09d75ab52373): 'Validate collection'  
info: WorkerServiceOptions.Example.Worker[1]  
      => Processing work, started at: 09/25/2020 14:30:45  
      Processing priority item: Priority-Medium (dea9e3f4-d7df-46d2-b7cd-5e0232eb98a5): 'Propagate selections'  
info: WorkerServiceOptions.Example.Worker[1]  
      => Processing work, started at: 09/25/2020 14:30:45  
      Processing priority item: Priority-Medium (089d7f0d-da72-4b55-92fe-57b147838056): 'Enter pooling [contention]'  
info: WorkerServiceOptions.Example.Worker[1]  
      => Processing work, started at: 09/25/2020 14:30:45  
      Processing priority item: Priority-Low (6e68c4be-089f-4450-9080-1ea63fcbb686): 'Health check network'  
info: WorkerServiceOptions.Example.Worker[1]  
      => Processing work, started at: 09/25/2020 14:30:45  
      Processing priority item: Priority-Deferred (6f324134-6bb6-455f-81d4-553ab307c421): 'Ping weather service'  
info: WorkerServiceOptions.Example.Worker[1]  
      => Processing work, started at: 09/25/2020 14:30:45  
      Processing priority item: Priority-Deferred (37bf736c-7a26-4a2a-9e56-e89bcf3b8f35): 'Set process state'
```

Log level guarded optimizations

Another performance optimization can be made by checking the [LogLevel](#), with `ILogger.IsEnabled(LogLevel)` before an invocation to the corresponding `Log*` method. When logging isn't configured for the given `LogLevel`, the following statements are true:

- `ILogger.Log` isn't called.
- An allocation of `object[]` representing the parameters is avoided.
- Value type boxing is avoided.

For more information:

- Micro benchmarks in the .NET runtime ↗
- Background and motivation for log level checks ↗

See also

- [Logging in .NET](#)

 [Collaborate with us on GitHub](#)

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

Console log formatting

Article • 06/23/2023

In .NET 5, support for custom formatting was added to console logs in the `Microsoft.Extensions.Logging.Console` namespace. There are three predefined formatting options available: [Simple](#), [Systemd](#), and [Json](#).

ⓘ Important

Previously, the `ConsoleLoggerFormat` enum allowed for selecting the desired log format, either human readable which was the `Default`, or single line which is also known as `Systemd`. However, these were **not** customizable, and are now deprecated.

In this article, you will learn about console log formatters. The sample source code demonstrates how to:

- Register a new formatter
- Select a registered formatter to use
 - Either through code, or [configuration](#)
- Implement a custom formatter
 - Update configuration via `IOptionsMonitor<TOptions>`
 - Enable custom color formatting

ⓘ Tip

All of the logging example source code is available in the [Samples Browser](#) for download. For more information, see [Browse code samples: Logging in .NET](#).

Register formatter

The [Console logging provider](#) has several predefined formatters, and exposes the ability to author your own custom formatter. To register any of the available formatters, use the corresponding `Add{Type}Console` extension method:

| Available types | Method to register type |
|--|---|
| ConsoleFormatterNames.Json | <code>ConsoleLoggerExtensions.AddJsonConsole</code> |

| Available types | Method to register type |
|---|---|
| ConsoleFormatterNames.Simple | ConsoleLoggerExtensions.AddSimpleConsole |
| ConsoleFormatterNames.Systemd | ConsoleLoggerExtensions.AddSystemdConsole |

Simple

To use the `Simple` console formatter, register it with `AddSimpleConsole`:

C#

```
using Microsoft.Extensions.Logging;

using ILoggerFactory loggerFactory =
    LoggerFactory.Create(builder =>
        builder.AddSimpleConsole(options =>
    {
        options.IncludeScopes = true;
        options.SingleLine = true;
        options.TimestampFormat = "HH:mm:ss ";
    }));
    
ILogger<Program> logger = loggerFactory.CreateLogger<Program>();
using (logger.BeginScope("[scope is enabled]"))
{
    logger.LogInformation("Hello World!");
    logger.LogInformation("Logs contain timestamp and log level.");
    logger.LogInformation("Each log message is fit in a single line.");
}
```

In the preceding sample source code, the `ConsoleFormatterNames.Simple` formatter was registered. It provides logs with the ability to not only wrap information such as time and log level in each log message, but also allows for ANSI color embedding and indentation of messages.

Systemd

The `ConsoleFormatterNames.Systemd` console logger:

- Uses the "Syslog" log level format and severities
- Does **not** format messages with colors
- Always logs messages in a single line

This is commonly useful for containers, which often make use of `Systemd` console logging. With .NET 5, the `Simple` console logger also enables a compact version that

logs in a single line, and also allows for disabling colors as shown in an earlier sample.

C#

```
using Microsoft.Extensions.Logging;

using ILoggerFactory loggerFactory =
    LoggerFactory.Create(builder =>
        builder.AddSystemdConsole(options =>
    {
        options.IncludeScopes = true;
        options.TimestampFormat = "HH:mm:ss ";
    }));
}

ILogger<Program> logger = loggerFactory.CreateLogger<Program>();
using (logger.BeginScope("[scope is enabled]"))
{
    logger.LogInformation("Hello World!");
    logger.LogInformation("Logs contain timestamp and log level.");
    logger.LogInformation("Systemd console logs never provide color
options.");
    logger.LogInformation("Systemd console logs always appear in a single
line.");
}
```

Json

To write logs in a JSON format, the `Json` console formatter is used. The sample source code shows how an ASP.NET Core app might register it. Using the `webapp` template, create a new ASP.NET Core app with the `dotnet new` command:

.NET CLI

```
dotnet new webapp -o Console.ExampleFormatters.Json
```

When running the app, using the template code, you get the default log format below:

Console

```
info: Console.ExampleFormatters.Json.Startup[0]
      Hello .NET friends!
info: Microsoft.Hosting.Lifetime[14]
      Now listening on: https://localhost:5001
info: Microsoft.Hosting.Lifetime[14]
      Now listening on: http://localhost:5000
info: Microsoft.Hosting.Lifetime[0]
      Application started. Press Ctrl+C to shut down.
info: Microsoft.Hosting.Lifetime[0]
      Hosting environment: Development
```

```
info: Microsoft.Hosting.Lifetime[0]
      Content root path: .\snippets\logging\console-formatter-json
```

By default, the `Simple` console log formatter is selected with default configuration. You can change this by calling `AddJsonConsole` in the `Program.cs`:

C#

```
using System.Text.Json;

HostApplicationBuilder builder = Host.CreateApplicationBuilder(args);

builder.Logging.AddJsonConsole(options =>
{
    options.IncludeScopes = false;
    options.TimestampFormat = "HH:mm:ss ";
    options.JsonWriterOptions = new JsonWriterOptions
    {
        Indented = true
    };
});

using IHost host = builder.Build();

var logger =
    host.Services
        .GetRequiredService<ILoggerFactory>()
        .CreateLogger<Program>();

logger.LogInformation("Hello .NET friends!");

await host.RunAsync();
```

Alternatively, you can also configure this using logging configuration, such as that found in the `appsettings.json` file:

JSON

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft": "Warning",
      "Microsoft.Hosting.Lifetime": "Information"
    },
    "Console": {
      "LogLevel": {
        "Default": "Information",
        "Microsoft": "Warning",
        "Microsoft.Hosting.Lifetime": "Information"
      },
    }
}
```

```
        "FormatterName": "json",
        "FormatterOptions": {
            "SingleLine": true,
            "IncludeScopes": true,
            "TimestampFormat": "HH:mm:ss",
            "UseUtcTimestamp": true,
            "JsonWriterOptions": {
                "Indented": true
            }
        }
    },
    "AllowedHosts": "*"
}
```

Run the app again, with the above change, the log message is now formatted as JSON:

JSON

```
{
    "Timestamp": "02:28:19",
    "EventId": 0,
    "LogLevel": "Information",
    "Category": "Console.ExampleFormatters.Json.Startup",
    "Message": "Hello .NET friends!",
    "State": {
        "Message": "Hello .NET friends!",
        "{OriginalFormat}": "Hello .NET friends!"
    }
}
{
    "Timestamp": "02:28:21",
    "EventId": 14,
    "LogLevel": "Information",
    "Category": "Microsoft.Hosting.Lifetime",
    "Message": "Now listening on: https://localhost:5001",
    "State": {
        "Message": "Now listening on: https://localhost:5001",
        "address": "https://localhost:5001",
        "{OriginalFormat}": "Now listening on: {address}"
    }
}
{
    "Timestamp": "02:28:21",
    "EventId": 14,
    "LogLevel": "Information",
    "Category": "Microsoft.Hosting.Lifetime",
    "Message": "Now listening on: http://localhost:5000",
    "State": {
        "Message": "Now listening on: http://localhost:5000",
        "address": "http://localhost:5000",
        "{OriginalFormat}": "Now listening on: {address}"
    }
}
```

```

}
{
  "Timestamp": "02:28:21 ",
  "EventId": 0,
  "LogLevel": "Information",
  "Category": "Microsoft.Hosting.Lifetime",
  "Message": "Application started. Press Ctrl+Shift+B to shut down.",
  "State": {
    "Message": "Application started. Press Ctrl+Shift+B to shut down.",
    "{OriginalFormat)": "Application started. Press Ctrl+Shift+B to shut
down."
  }
}
{
  "Timestamp": "02:28:21 ",
  "EventId": 0,
  "LogLevel": "Information",
  "Category": "Microsoft.Hosting.Lifetime",
  "Message": "Hosting environment: Development",
  "State": {
    "Message": "Hosting environment: Development",
    "envName": "Development",
    "{OriginalFormat)": "Hosting environment: {envName}"
  }
}
{
  "Timestamp": "02:28:21 ",
  "EventId": 0,
  "LogLevel": "Information",
  "Category": "Microsoft.Hosting.Lifetime",
  "Message": "Content root path: .\\snippets\\logging\\console-formatter-
json",
  "State": {
    "Message": "Content root path: .\\snippets\\logging\\console-formatter-
json",
    "contentRoot": ".\\snippets\\logging\\console-formatter-json",
    "{OriginalFormat)": "Content root path: {contentRoot}"
  }
}

```

💡 Tip

The `Json` console formatter, by default, logs each message in a single line. In order to make it more readable while configuring the formatter, set `JsonWriterOptions.Indented` to `true`.

⊗ Caution

When using the Json console formatter, do not pass in log messages that have already been serialized as JSON. The logging infrastructure itself already manages the serialization of log messages, so if you're to pass in a log message that is already serialized—it will be double serialized, thus causing malformed output.

Set formatter with configuration

The previous samples have shown how to register a formatter programmatically. Alternatively, this can be done with [configuration](#). Consider the previous web application sample source code, if you update the `appsettings.json` file rather than calling `ConfigureLogging` in the `Program.cs` file, you could get the same outcome. The updated `appsettings.json` file would configure the formatter as follows:

JSON

```
{  
  "Logging": {  
    "LogLevel": {  
      "Default": "Information",  
      "Microsoft": "Warning",  
      "Microsoft.Hosting.Lifetime": "Information"  
    },  
    "Console": {  
      "LogLevel": {  
        "Default": "Information",  
        "Microsoft": "Warning",  
        "Microsoft.Hosting.Lifetime": "Information"  
      },  
      "FormatterName": "json",  
      "FormatterOptions": {  
        "SingleLine": true,  
        "IncludeScopes": true,  
        "TimestampFormat": "HH:mm:ss ",  
        "UseUtcTimestamp": true,  
        "JsonWriterOptions": {  
          "Indented": true  
        }  
      }  
    },  
    "AllowedHosts": "*"  
  }  
}
```

The two key values that need to be set are `"FormatterName"` and `"FormatterOptions"`. If a formatter with the value set for `"FormatterName"` is already registered, that formatter is selected, and its properties can be configured as long as they are provided as a key

inside the `"FormatterOptions"` node. The predefined formatter names are reserved under [ConsoleFormatterNames](#):

- [ConsoleFormatterNames.Json](#)
- [ConsoleFormatterNames.Simple](#)
- [ConsoleFormatterNames.Systemd](#)

Implement a custom formatter

To implement a custom formatter, you need to:

- Create a subclass of [ConsoleFormatter](#), this represents your custom formatter
- Register your custom formatter with
 - [ConsoleLoggerExtensions.AddConsole](#)
 - [ConsoleLoggerExtensions.AddConsoleFormatter<TFormatter,TOptions>\(ILoggingBuilder, Action<TOptions>\)](#)

Create an extension method to handle this for you:

C#

```
using Microsoft.Extensions.Logging;

namespace Console.ExampleFormatters.Custom;

public static class ConsoleLoggerExtensions
{
    public static ILoggingBuilder AddCustomFormatter(
        this ILoggingBuilder builder,
        Action<CustomOptions> configure) =>
        builder.AddConsole(options => options.FormatterName = "customName")
            .AddConsoleFormatter<CustomFormatter, CustomOptions>(configure);
}
```

The `CustomOptions` are defined as follows:

C#

```
using Microsoft.Extensions.Logging.Console;

namespace Console.ExampleFormatters.Custom;

public sealed class CustomOptions : ConsoleFormatterOptions
{
    public string? CustomPrefix { get; set; }
}
```

In the preceding code, the options are a subclass of `ConsoleFormatterOptions`.

The `AddConsoleFormatter` API:

- Registers a subclass of `ConsoleFormatter`
- Handles configuration:
 - Uses a change token to synchronize updates, based on the [options pattern](#), and the `IOptionsMonitor` interface

C#

```
using Console.ExampleFormatters.Custom;
using Microsoft.Extensions.Logging;

using ILoggerFactory loggerFactory =
    LoggerFactory.Create(builder =>
        builder.AddCustomFormatter(options =>
            options.CustomPrefix = " ~~~~~ "));

ILogger<Program> logger = loggerFactory.CreateLogger<Program>();
using (logger.BeginScope("TODO: Add logic to enable scopes"))
{
    logger.LogInformation("Hello World!");
    logger.LogInformation("TODO: Add logic to enable timestamp and log level
info.");
}
```

Define a `CustomFormatter` subclass of `ConsoleFormatter`:

C#

```
using Microsoft.Extensions.Logging;
using Microsoft.Extensions.Logging.Abstractions;
using Microsoft.Extensions.Logging.Console;
using Microsoft.Extensions.Options;

namespace Console.ExampleFormatters.Custom;

public sealed class CustomFormatter : ConsoleFormatter, IDisposable
{
    private readonly IDisposable? _optionsReloadToken;
    private CustomOptions _formatterOptions;

    public CustomFormatter(IOptionsMonitor<CustomOptions> options)
        // Case insensitive
        : base("customName") =>
            (_optionsReloadToken, _formatterOptions) =
                (options.OnChange(ReloadLoggerOptions), options.CurrentValue);

    private void ReloadLoggerOptions(CustomOptions options) =>
        _formatterOptions = options;
```

```

public override void Write<TState>(
    in LogEntry<TState> logEntry,
    IExternalScopeProvider? scopeProvider,
    TextWriter textWriter)
{
    string? message =
        logEntry.Formatter?.Invoke(
            logEntry.State, logEntry.Exception);

    if (message is null)
    {
        return;
    }

    CustomLogicGoesHere(textWriter);
    textWriter.WriteLine(message);
}

private void CustomLogicGoesHere(TextWriter textWriter)
{
    textWriter.Write(_formatterOptions.CustomPrefix);
}

public void Dispose() => _optionsReloadToken?.Dispose();
}

```

The preceding `CustomFormatter.Write<TState>` API dictates what text gets wrapped around each log message. A standard `ConsoleFormatter` should be able to wrap around scopes, time stamps, and severity level of logs at a minimum. Additionally, you can encode ANSI colors in the log messages, and provide desired indentations as well. The implementation of the `CustomFormatter.Write<TState>` lacks these capabilities.

For inspiration on further customizing formatting, see the existing implementations in the `Microsoft.Extensions.Logging.Console` namespace:

- [SimpleConsoleFormatter](#).
- [SystemdConsoleFormatter](#)
- [JsonConsoleFormatter](#)

Custom configuration options

To further customize the logging extensibility, your derived `ConsoleFormatterOptions` class can be configured from any [configuration provider](#). For example, you could use the [JSON configuration provider](#) to define your custom options. First define your `ConsoleFormatterOptions` subclass.

C#

```
using Microsoft.Extensions.Logging.Console;

namespace Console.ExampleFormatters.CustomWithConfig;

public sealed class CustomWrappingConsoleFormatterOptions : 
ConsoleFormatterOptions
{
    public string? CustomPrefix { get; set; }

    public string? CustomSuffix { get; set; }
}
```

The preceding console formatter options class defines two custom properties, representing a prefix and suffix. Next, define the *appsettings.json* file that will configure your console formatter options.

JSON

```
{
    "Logging": {
        "LogLevel": {
            "Default": "Information",
            "Microsoft": "Warning",
            "Microsoft.Hosting.Lifetime": "Information"
        },
        "Console": {
            "LogLevel": {
                "Default": "Information",
                "Microsoft": "Warning",
                "Microsoft.Hosting.Lifetime": "Information"
            },
            "FormatterName": "CustomTimePrefixingFormatter",
            "FormatterOptions": {
                "CustomPrefix": "|-<[",
                "CustomSuffix": "]>-|",
                "SingleLine": true,
                "IncludeScopes": true,
                "TimestampFormat": "HH:mm:ss.fffff ",
                "UseUtcTimestamp": true,
                "JsonWriterOptions": {
                    "Indented": true
                }
            }
        }
    },
    "AllowedHosts": "*"
}
```

In the preceding JSON config file:

- The "Logging" node defines a "Console".
- The "Console" node specifies a "FormatterName" of "CustomTimePrefixingFormatter", which maps to a custom formatter.
- The "FormatterOptions" node defines a "CustomPrefix", and "CustomSuffix", as well as a few other derived options.

💡 Tip

The `$.Logging.Console.FormatterOptions` JSON path is reserved, and will map to a custom **ConsoleFormatterOptions** when added using the **AddConsoleFormatter** extension method. This provides the ability to define custom properties, in addition to the ones available.

Consider the following `CustomDatePrefixingFormatter`:

C#

```
using Microsoft.Extensions.Logging;
using Microsoft.Extensions.Logging.Abstractions;
using Microsoft.Extensions.Logging.Console;
using Microsoft.Extensions.Options;

namespace Console.ExampleFormatters.CustomWithConfig;

public sealed class CustomTimePrefixingFormatter : ConsoleFormatter,
IDisposable
{
    private readonly IDisposable? _optionsReloadToken;
    private CustomWrappingConsoleFormatterOptions _formatterOptions;

    public CustomTimePrefixingFormatter(IOptionsMonitor<CustomWrappingConsoleFormatterOptions> options)
        // Case insensitive
        : base(nameof(CustomTimePrefixingFormatter)) =>
        (_optionsReloadToken, _formatterOptions) =
            (options.OnChange(ReloadLoggerOptions), options.CurrentValue);

    private void ReloadLoggerOptions(CustomWrappingConsoleFormatterOptions options) =>
        _formatterOptions = options;

    public override void Write<TState>(
        in LogEntry<TState> logEntry,
        IExternalScopeProvider? scopeProvider,
        TextWriter textWriter)
    {
        string message =
            logEntry.Formatter(
```

```

        logEntry.State, logEntry.Exception);

    if (message == null)
    {
        return;
    }

    WritePrefix(textWriter);
    textWriter.Write(message);
    WriteSuffix(textWriter);
}

private void WritePrefix(TextWriter textWriter)
{
    DateTime now = _formatterOptions.UseUtcTimestamp
        ? DateTime.UtcNow
        : DateTime.Now;

    textWriter.WriteLine($""""
        {_formatterOptions.CustomPrefix}
{now.ToString(_formatterOptions.TimestampFormat)}
""");
}

private void WriteSuffix(TextWriter textWriter) =>
    textWriter.WriteLine($" {_formatterOptions.CustomSuffix}");
}

public void Dispose() => _optionsReloadToken?.Dispose();
}

```

In the preceding formatter implementation:

- The `CustomWrappingConsoleFormatterOptions` are monitored for change, and updated accordingly.
- Messages that are written are wrapped with the configured prefix, and suffix.
- A timestamp is added after the prefix, but before the message using the configured `ConsoleFormatterOptions.UseUtcTimestamp` and `ConsoleFormatterOptions.TimestampFormat` values.

To use custom configuration options, with custom formatter implementations, add when calling `ConfigureLogging(IHostBuilder, Action<HostBuilderContext, ILoggingBuilder>)`.

C#

```

using Console.ExampleFormatters.CustomWithConfig;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
using Microsoft.Extensions.Logging;

HostApplicationBuilder builder = Host.CreateApplicationBuilder(args);

```

```
builder.Logging.AddConsole()
    .AddConsoleFormatter<
        CustomTimePrefixingFormatter, CustomWrappingConsoleFormatterOptions>
();
using IHost host = builder.Build();

ILoggerFactory loggerFactory =
host.Services.GetRequiredService<ILoggerFactory>();
	ILogger<Program> logger = loggerFactory.CreateLogger<Program>();

using (logger.BeginScope("Logging scope"))
{
    logger.LogInformation("Hello World!");
    logger.LogInformation("The .NET developer community happily welcomes
you.");
}
```

The following console output is similar to what you might expect to see from using this `CustomTimePrefixingFormatter`.

Console

```
|-<[ 15:03:15.6179 Hello World! ]>-|
|-<[ 15:03:15.6347 The .NET developer community happily welcomes you. ]>-|
```

Implement custom color formatting

In order to properly enable color capabilities in your custom logging formatter, you can extend the `SimpleConsoleFormatterOptions` as it has a `SimpleConsoleFormatterOptions.ColorBehavior` property that can be useful for enabling colors in logs.

Create a `CustomColorOptions` that derives from `SimpleConsoleFormatterOptions`:

C#

```
using Microsoft.Extensions.Logging.Console;

namespace Console.ExampleFormatters.Custom;

public class CustomColorOptions : SimpleConsoleFormatterOptions
{
    public string? CustomPrefix { get; set; }
}
```

Next, write some extension methods in a `TextWriterExtensions` class that allow for conveniently embedding ANSI coded colors within formatted log messages:

C#

```
namespace Console.ExampleFormatters.Custom;

public static class TextWriterExtensions
{
    const string DefaultForegroundColor = "\x1B[39m\x1B[22m";
    const string DefaultBackgroundColor = "\x1B[49m";

    public static void WriteWithColor(
        this TextWriter textWriter,
        string message,
        ConsoleColor? background,
        ConsoleColor? foreground)
    {
        // Order:
        // 1. background color
        // 2. foreground color
        // 3. message
        // 4. reset foreground color
        // 5. reset background color

        var backgroundColor = background.HasValue ?
GetBackgroundColorEscapeCode(background.Value) : null;
        var foregroundColor = foreground.HasValue ?
GetForegroundColorEscapeCode(foreground.Value) : null;

        if (backgroundColor != null)
        {
            textWriter.Write(backgroundColor);
        }
        if (foregroundColor != null)
        {
            textWriter.Write(foregroundColor);
        }

        textWriter.WriteLine(message);

        if (foregroundColor != null)
        {
            textWriter.Write(DefaultForegroundColor);
        }
        if (backgroundColor != null)
        {
            textWriter.Write(DefaultBackgroundColor);
        }
    }

    static string GetForegroundColorEscapeCode(ConsoleColor color) =>
        color switch
    {
        ConsoleColor.Black => "\x1B[30m",
        ConsoleColor.DarkRed => "\x1B[31m",
        ConsoleColor.DarkGreen => "\x1B[32m",
    }
}
```

```

        ConsoleColor.DarkYellow => "\x1B[33m",
        ConsoleColor.DarkBlue => "\x1B[34m",
        ConsoleColor.DarkMagenta => "\x1B[35m",
        ConsoleColor.DarkCyan => "\x1B[36m",
        ConsoleColor.Gray => "\x1B[37m",
        ConsoleColor.Red => "\x1B[1m\x1B[31m",
        ConsoleColor.Green => "\x1B[1m\x1B[32m",
        ConsoleColor.Yellow => "\x1B[1m\x1B[33m",
        ConsoleColor.Blue => "\x1B[1m\x1B[34m",
        ConsoleColor.Magenta => "\x1B[1m\x1B[35m",
        ConsoleColor.Cyan => "\x1B[1m\x1B[36m",
        ConsoleColor.White => "\x1B[1m\x1B[37m",

        _ => DefaultForegroundColor
    };

    static string GetBackgroundColorEscapeCode(ConsoleColor color) =>
        color switch
    {
        ConsoleColor.Black => "\x1B[40m",
        ConsoleColor.DarkRed => "\x1B[41m",
        ConsoleColor.DarkGreen => "\x1B[42m",
        ConsoleColor.DarkYellow => "\x1B[43m",
        ConsoleColor.DarkBlue => "\x1B[44m",
        ConsoleColor.DarkMagenta => "\x1B[45m",
        ConsoleColor.DarkCyan => "\x1B[46m",
        ConsoleColor.Gray => "\x1B[47m",

        _ => DefaultBackgroundColor
    };
}

```

A custom color formatter that handles applying custom colors could be defined as follows:

```

C#

using Microsoft.Extensions.Logging;
using Microsoft.Extensions.Logging.Abstractions;
using Microsoft.Extensions.Logging.Console;
using Microsoft.Extensions.Options;

namespace Console.ExampleFormatters.Custom;

public sealed class CustomColorFormatter : ConsoleFormatter, IDisposable
{
    private readonly IDisposable? _optionsReloadToken;
    private CustomColorOptions _formatterOptions;

    private bool ConsoleColorFormattingEnabled =>
        _formatterOptions.ColorBehavior == LoggerColorBehavior.Enabled ||
        _formatterOptions.ColorBehavior == LoggerColorBehavior.Default &&
        System.Console.IsOutputRedirected == false;

```

```

public CustomColorFormatter(IOptionsMonitor<CustomColorOptions> options)
    // Case insensitive
    : base("customName") =>
    (_optionsReloadToken, _formatterOptions) =
        (options.OnChange(ReloadLoggerOptions), options.CurrentValue);

private void ReloadLoggerOptions(CustomColorOptions options) =>
    _formatterOptions = options;

public override void Write<TState>(
    in LogEntry<TState> logEntry,
    IExternalScopeProvider? scopeProvider,
    TextWriter textWriter)
{
    if (logEntry.Exception is null)
    {
        return;
    }

    string? message =
        logEntry.Formatter?.Invoke(
            logEntry.State, logEntry.Exception);

    if (message is null)
    {
        return;
    }

    CustomLogicGoesHere(textWriter);
    textWriter.WriteLine(message);
}

private void CustomLogicGoesHere(TextWriter textWriter)
{
    if (ConsoleColorFormattingEnabled)
    {
        textWriter.WriteLineColor(
            _formatterOptions.CustomPrefix ?? string.Empty,
            ConsoleColor.Black,
            ConsoleColor.Green);
    }
    else
    {
        textWriter.Write(_formatterOptions.CustomPrefix);
    }
}

public void Dispose() => _optionsReloadToken?.Dispose();
}

```

When you run the application, the logs will show the `CustomPrefix` message in the color green when `FormatterOptions.ColorBehavior` is `Enabled`.

Note

When `LoggerColorBehavior` is `Disabled`, log messages do *not* interpret embedded ANSI color codes within log messages. Instead, they output the raw message. For example, consider the following:

C#

```
logger.LogInformation("Random log \x1B[42mwith green background\x1B[49m message");
```

This would output the verbatim string, and it is *not* colorized.

Output

```
Random log \x1B[42mwith green background\x1B[49m message
```

See also

- [Logging in .NET](#)
- [Implement a custom logging provider in .NET](#)
- [High-performance logging in .NET](#)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

.NET Generic Host

Article • 12/13/2023

In this article, you learn about the various patterns for configuring and building a .NET Generic Host available in the [Microsoft.Extensions.Hosting](#) NuGet package. The .NET Generic Host is responsible for app startup and lifetime management. The Worker Service templates create a .NET Generic Host, [HostApplicationBuilder](#). The Generic Host can be used with other types of .NET applications, such as Console apps.

A *host* is an object that encapsulates an app's resources and lifetime functionality, such as:

- Dependency injection (DI)
- Logging
- Configuration
- App shutdown
- `IHostedService` implementations

When a host starts, it calls `IHostedService.StartAsync` on each implementation of `IHostedService` registered in the service container's collection of hosted services. In a worker service app, all `IHostedService` implementations that contain `BackgroundService` instances have their `BackgroundService.ExecuteAsync` methods called.

The main reason for including all of the app's interdependent resources in one object is lifetime management: control over app startup and graceful shutdown.

Set up a host

The host is typically configured, built, and run by code in the `Program` class. The `Main` method:

`IHostApplicationBuilder`

- Calls a `CreateApplicationBuilder` method to create and configure a builder object.
- Calls `Build()` to create an `IHost` instance.
- Calls `Run` or `RunAsync` method on the host object.

The .NET Worker Service templates generate the following code to create a Generic Host:

C#

```
using Example.WorkerService;

HostApplicationBuilder builder = Host.CreateApplicationBuilder(args);
builder.Services.AddHostedService<Worker>();

IHost host = builder.Build();
host.Run();
```

For more information on Worker Services, see [Worker Services in .NET](#).

Host builder settings

IHostApplicationBuilder

The [CreateApplicationBuilder](#) method:

- Sets the content root to the path returned by [GetCurrentDirectory\(\)](#).
- Loads [host configuration](#) from:
 - Environment variables prefixed with `DOTNET_`.
 - Command-line arguments.
- Loads app configuration from:
 - `appsettings.json`.
 - `appsettings.{Environment}.json`.
 - Secret Manager when the app runs in the `Development` environment.
 - Environment variables.
 - Command-line arguments.
- Adds the following logging providers:
 - Console
 - Debug
 - EventSource
 - EventLog (only when running on Windows)
- Enables scope validation and [dependency validation](#) when the environment is `Development`.

The [HostApplicationBuilder.Services](#) is an [Microsoft.Extensions.DependencyInjection.IServiceCollection](#) instance. These services are used to build an [IServiceProvider](#) that's used with dependency injection to resolve the registered services.

Framework-provided services

When you call either [IHostBuilder.Build\(\)](#) or [HostApplicationBuilder.Build\(\)](#), the following services are registered automatically:

- [IHostApplicationLifetime](#)
- [IHostLifetime](#)
- [IHostEnvironment](#)

IHostApplicationLifetime

Inject the [IHostApplicationLifetime](#) service into any class to handle post-startup and graceful shutdown tasks. Three properties on the interface are cancellation tokens used to register app start and app stop event handler methods. The interface also includes a [StopApplication\(\)](#) method.

The following example is an [IHostedService](#) implementation that registers [IHostApplicationLifetime](#) events:

C#

```
using Microsoft.Extensions.Hosting;
using Microsoft.Extensions.Logging;

namespace AppLifetime.Example;

public sealed class ExampleHostedService : IHostedService
{
    private readonly ILogger _logger;

    public ExampleHostedService(
        ILogger<ExampleHostedService> logger,
        IHostApplicationLifetime appLifetime)
    {
        _logger = logger;

        appLifetime.ApplicationStarted.Register(OnStarted);
        appLifetime.ApplicationStopping.Register(OnStopping);
        appLifetime.ApplicationStopped.Register(OnStopped);
    }

    public Task StartAsync(CancellationToken cancellationToken)
    {
        _logger.LogInformation("1. StartAsync has been called.");

        return Task.CompletedTask;
    }
}
```

```
public Task StopAsync(CancellationToken cancellationToken)
{
    _logger.LogInformation("4. StopAsync has been called.");

    return Task.CompletedTask;
}

private void OnStarted()
{
    _logger.LogInformation("2. OnStarted has been called.");
}

private void OnStopping()
{
    _logger.LogInformation("3. OnStopping has been called.");
}

private void OnStopped()
{
    _logger.LogInformation("5. OnStopped has been called.");
}
}
```

The Worker Service template could be modified to add the `ExampleHostedService` implementation:

C#

```
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
using AppLifetime.Example;

HostApplicationBuilder builder = Host.CreateApplicationBuilder(args);

builder.Services.AddHostedService<ExampleHostedService>();
using IHost host = builder.Build();

await host.RunAsync();
```

The application would write the following sample output:

C#

```
// Sample output:
//      info: ExampleHostedService[0]
//          1. StartAsync has been called.
//      info: ExampleHostedService[0]
//          2. OnStarted has been called.
//      info: Microsoft.Hosting.Lifetime[0]
//          Application started. Press Ctrl+C to shut down.
//      info: Microsoft.Hosting.Lifetime[0]
```

```
//      Hosting environment: Production
//      info: Microsoft.Hosting.Lifetime[0]
//      Content root path: ..\app-lifetime\bin\Debug\net7.0
//      info: ExampleHostedService[0]
//      3. OnStopping has been called.
//      info: Microsoft.Hosting.Lifetime[0]
//      Application is shutting down...
//      info: ExampleHostedService[0]
//      4. StopAsync has been called.
//      info: ExampleHostedService[0]
//      5. OnStopped has been called.
```

IHostLifetime

The [IHostLifetime](#) implementation controls when the host starts and when it stops. The last implementation registered is used.

`Microsoft.Extensions.Hosting.Internal.ConsoleLifetime` is the default `IHostLifetime` implementation. For more information on the lifetime mechanics of shutdown, see [Host shutdown](#).

IHostEnvironment

Inject the [IHostEnvironment](#) service into a class to get information about the following settings:

- [IHostEnvironment.ApplicationName](#)
- [IHostEnvironment.ContentRootFileProvider](#)
- [IHostEnvironment.ContentRootPath](#)
- [IHostEnvironment.EnvironmentName](#)

Additionally, the `IHostEnvironment` service exposes the ability to evaluate the environment with the help of these extension methods:

- [HostingEnvironmentExtensions.IsDevelopment](#)
- [HostingEnvironmentExtensions.IsEnvironment](#)
- [HostingEnvironmentExtensions.IsProduction](#)
- [HostingEnvironmentExtensions.IsStaging](#)

Host configuration

Host configuration is used to configure properties of the [IHostEnvironment](#) implementation.

IHostApplicationBuilder

The host configuration is available in [IHostApplicationBuilder.Configuration](#) property and the environment implementation is available in [IHostApplicationBuilder.Environment](#) property. To configure the host, access the [Configuration](#) property and call any of the available extension methods.

To add host configuration, consider the following example:

C#

```
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.Hosting;

HostApplicationBuilder builder = Host.CreateApplicationBuilder(args);

builder.Environment.ContentRootPath = Directory.GetCurrentDirectory();
builder.Configuration.AddJsonFile("hostsettings.json", optional: true);
builder.Configuration.AddEnvironmentVariables(prefix: "PREFIX_");
builder.Configuration.AddCommandLine(args);

using IHost host = builder.Build();

// Application code should start here.

await host.RunAsync();
```

The preceding code:

- Sets the content root to the path returned by [GetCurrentDirectory\(\)](#).
- Loads host configuration from:
 - *hostsettings.json*.
 - Environment variables prefixed with `PREFIX_`.
 - Command-line arguments.

App configuration

IHostApplicationBuilder

App configuration is created by calling [ConfigureAppConfiguration](#) on an [IHostApplicationBuilder](#). The public [IHostApplicationBuilder.Configuration](#) property

allows consumers to read from or make changes to the existing configuration using available extension methods.

For more information, see [Configuration in .NET](#).

Host shutdown

There are several ways in which a hosted process is stopped. Most commonly, a hosted process can be stopped in the following ways:

- If someone doesn't call [Run](#) or [HostingAbstractionsHostExtensions.WaitForShutdown](#) and the app exits normally with `Main` completing.
- If the app crashes.
- If the app is forcefully shut down using [SIGKILL](#) (or `CTRL + Z`).

The hosting code isn't responsible for handling these scenarios. The owner of the process needs to deal with them the same as any other app. There are several other ways in which a hosted service process can be stopped:

- If `ConsoleLifetime` is used ([UseConsoleLifetime](#)), it listens for the following signals and attempts to stop the host gracefully.
 - [SIGINT](#) (or `CTRL + C`).
 - [SIGQUIT](#) (or `CTRL + BREAK` on Windows, `CTRL + \` on Unix).
 - [SIGTERM](#) (sent by other apps, such as `docker stop`).
- If the app calls [Environment.Exit](#).

The built-in hosting logic handles these scenarios, specifically the `ConsoleLifetime` class. `ConsoleLifetime` tries to handle the "shutdown" signals SIGINT, SIGQUIT, and SIGTERM to allow for a graceful exit to the application.

Before .NET 6, there wasn't a way for .NET code to gracefully handle SIGTERM. To work around this limitation, `ConsoleLifetime` would subscribe to [System.AppDomain.ProcessExit](#). When `ProcessExit` was raised, `ConsoleLifetime` would signal the host to stop and block the `ProcessExit` thread, waiting for the host to stop.

The process exit handler would allow for the clean-up code in the application to run—for example, [IHost.StopAsync](#) and code after [HostingAbstractionsHostExtensions.Run](#) in the `Main` method.

However, there were other issues with this approach because SIGTERM wasn't the only way `ProcessExit` was raised. SIGTERM is also raised when app code calls

`Environment.Exit`. `Environment.Exit` isn't a graceful way of shutting down a process in the `Microsoft.Extensions.Hosting` app model. It raises the `ProcessExit` event and then exits the process. The end of the `Main` method doesn't get executed. Background and foreground threads are terminated, and `finally` blocks *aren't* executed.

Since `ConsoleLifetime` blocked `ProcessExit` while waiting for the host to shut down, this behavior led to [deadlocks](#) from `Environment.Exit` also blocks waiting for the call to `ProcessExit`. Additionally, since the SIGTERM handling was attempting to gracefully shut down the process, `ConsoleLifetime` would set the `ExitCode` to `0`, which [clobbered](#) the user's exit code passed to `Environment.Exit`.

In .NET 6, [POSIX signals](#) are supported and handled. The `ConsoleLifetime` handles SIGTERM gracefully, and no longer gets involved when `Environment.Exit` is invoked.

💡 Tip

For .NET 6+, `ConsoleLifetime` no longer has logic to handle scenario `Environment.Exit`. Apps that call `Environment.Exit` and need to perform clean-up logic can subscribe to `ProcessExit` themselves. Hosting will no longer attempt to gracefully stop the host in these scenarios.

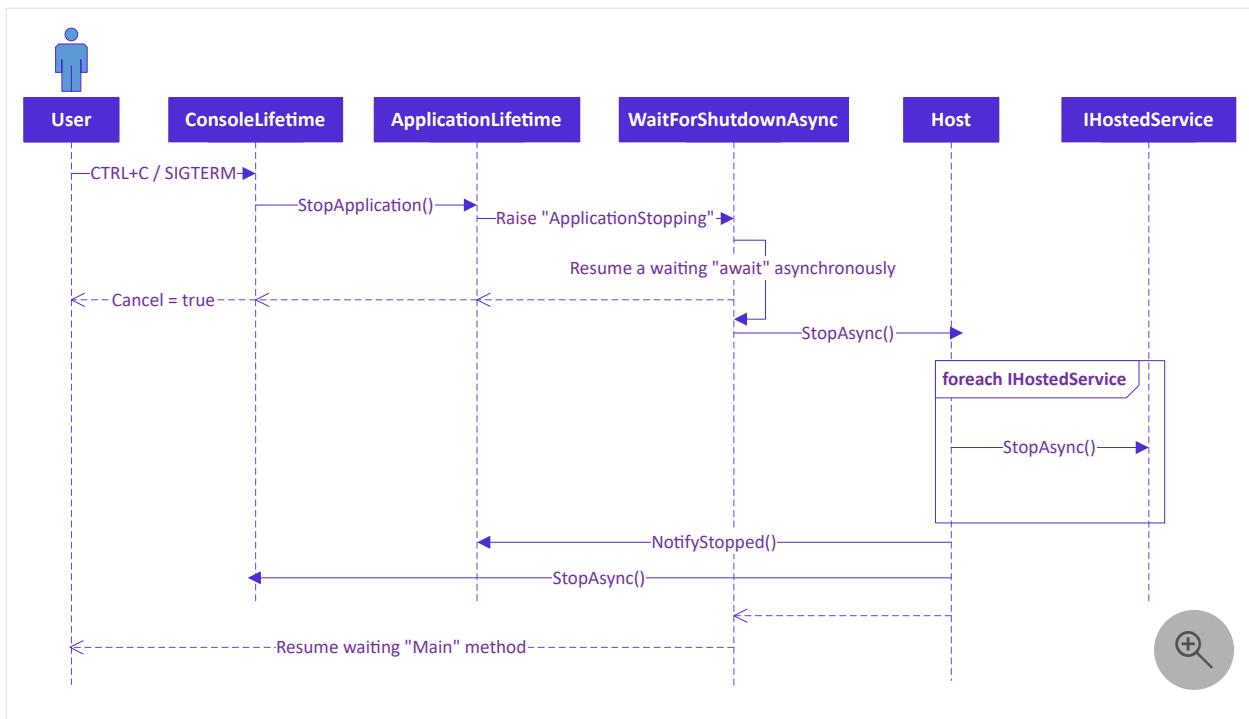
If your application uses hosting, and you want to gracefully stop the host, you can call `IHostApplicationLifetime.StopApplication` instead of `Environment.Exit`.

Hosting shutdown process

The following sequence diagram shows how the signals are handled internally in the hosting code. Most users don't need to understand this process. But for developers that need a deep understanding, a good visual may help you get started.

After the host has been started, when a user calls `Run` or `WaitForShutdown`, a handler gets registered for `IApplicationLifetime.ApplicationStopping`. Execution is paused in `WaitForShutdown`, waiting for the `ApplicationStopping` event to be raised. The `Main` method doesn't return right away, and the app stays running until `Run` or `WaitForShutdown` returns.

When a signal is sent to the process, it initiates the following sequence:



1. The control flows from `ConsoleLifetime` to `ApplicationLifetime` to raise the `ApplicationStopping` event. This signals `WaitForShutdownAsync` to unblock the `Main` execution code. In the meantime, the POSIX signal handler returns with `Cancel = true` since the POSIX signal has been handled.
2. The `Main` execution code starts executing again and tells the host to `StopAsync()`, which in turn stops all the hosted services, and raises any other stopped events.
3. Finally, `WaitForShutdown` exits, allowing for any application clean-up code to execute, and for the `Main` method to exit gracefully.

Host shutdown in web server scenarios

There are various other common scenarios in which graceful shutdown works in Kestrel for both HTTP/1.1 and HTTP/2 protocols, and how you can configure it in different environments with a load balancer to drain traffic smoothly. While web server configuration is beyond the scope of this article, you can find more information on [Configure options for the ASP.NET Core Kestrel web server documentation](#).

When the Host receives a shutdown signal (for example, `CTL + C` or `StopAsync`), it notifies the application by signaling `ApplicationStopping`. You should subscribe to this event if you have any long-running operations that need to finish gracefully.

Next, the Host calls `IHostBuilder`'s `StopAsync` with a shutdown timeout that you can configure (default 30s). Kestrel (and Http.Sys) close their port bindings and stop accepting new connections. They also tell the current connections to stop processing new requests. For HTTP/2 and HTTP/3, a preliminary `GOAWAY` message is sent to the client. For HTTP/1.1,

they stop the connection loop because requests are processed in order. IIS behaves differently, by rejecting new requests with a 503 status code.

The active requests have until the shutdown timeout to complete. If they're all complete before the timeout, the server returns control to the host sooner. If the timeout expires, the pending connections and requests are aborted forcefully, which can cause errors in the logs and to the clients.

Load balancer considerations

To ensure a smooth transition of clients to a new destination when working with a load balancer, you can follow these steps:

- Bring up the new instance and start balancing traffic to it (you may already have several instances for scaling purposes).
- Disable or remove the old instance in the load balancer configuration so it stops receiving new traffic.
- Signal the old instance to shut down.
- Wait for it to drain or time out.

See also

- [Dependency injection in .NET](#)
- [Logging in .NET](#)
- [Configuration in .NET](#)
- [Worker Services in .NET](#)
- [ASP.NET Core Web Host](#)
- [ASP.NET Core Kestrel web server configuration](#)
- Generic host bugs should be created in the github.com/dotnet/runtime repo

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

.NET is an open source project.
Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

Introduction to resilient app development

Article • 11/29/2023

Resiliency is the ability of an app to recover from transient failures and continue to function. In the context of .NET programming, resilience is achieved by designing apps that can handle failures gracefully and recover quickly. To help build resilient apps in .NET, the following two packages are available on NuGet:

[+] Expand table

| NuGet package | Description |
|--|--|
|  Microsoft.Extensions.Resilience | This NuGet package provides mechanisms to harden apps against transient failures. |
|  Microsoft.Extensions.Http.Resilience | This NuGet package provides resilience mechanisms specifically for the HttpClient class. |

These two NuGet packages are built on top of [Polly](#), which is a popular open-source project. Polly is a .NET resilience and transient fault-handling library that allows developers to express strategies such as [retry](#), [circuit breaker](#), timeout, [bulkhead isolation](#), [rate-limiting](#), fallback, and hedging in a fluent and thread-safe manner.

ⓘ Important

The [Microsoft.Extensions.Http.Polly](#) NuGet package is deprecated. Use either of the aforementioned packages instead.

Get started

To get started with resilience in .NET, install the [Microsoft.Extensions.Resilience](#) NuGet package.

.NET CLI

.NET CLI

```
dotnet add package Microsoft.Extensions.Resilience --version 8.0.0
```

For more information, see [dotnet add package](#) or [Manage package dependencies in .NET applications](#).

Build a resilience pipeline

To use resilience, you must first build a pipeline of resilience-based strategies. Each configured strategy executes in order of configuration. In other words, order is important. The entry point is an extension method on the [IServiceCollection](#) type, named `AddResiliencePipeline`. This method takes an identifier of the pipeline and a delegate that configures the pipeline. The delegate is passed an instance of `ResiliencePipelineBuilder`, which is used to add resilience strategies to the pipeline.

Consider the following string-based `key` example:

C#

```
using Microsoft.Extensions.DependencyInjection;
using Polly;
using Polly.CircuitBreaker;
using Polly.Registry;
using Polly.Retry;
using Polly.Timeout;

var services = new ServiceCollection();

const string key = "Retry-Timeout";

services.AddResiliencePipeline(key, static builder =>
{
    // See: https://www.pollydocs.org:strategies/retry.html
    builder.AddRetry(new RetryStrategyOptions
    {
        ShouldHandle = new
        PredicateBuilder().Handle<TimeoutRejectedException>()
    });

    // See: https://www.pollydocs.org:strategies/timeout.html
    builder.AddTimeout(TimeSpan.FromSeconds(1.5));
});
```

The preceding code:

- Creates a new `ServiceCollection` instance.
- Defines a `key` to identify the pipeline.
- Adds a resilience pipeline to the `ServiceCollection` instance.
- Configures the pipeline with a retry and timeout strategies.

Each pipeline is configured for a given `key`, and each `key` is used to identify its corresponding `ResiliencePipeline` when getting the pipeline from the provider. The `key` is a generic type parameter of the `AddResiliencePipeline` method.

Resilience pipeline builder extensions

To add a strategy to the pipeline, call any of the available `Add*` extension methods on the `ResiliencePipelineBuilder` instance.

- `AddRetry`: Try again if something fails, which is useful when the problem is temporary and might go away.
- `AddCircuitBreaker`: Stop trying if something is broken or busy, which benefits you by avoiding wasted time and making things worse.
- `AddTimeout`: Give up if something takes too long, which can improve performance by freeing up resources.
- `AddRateLimiter`: Limit how many requests you accept, which enables you to control inbound load.
- `AddConcurrencyLimiter`: Limit how many requests you make, which enables you to control outbound load.
- `AddFallback`: Do something else when experiencing failures, which improves user experience.
- `AddHedging`: Issue multiple requests in case of high latency or failure, which can improve responsiveness.

For more information, see [Resilience strategies](#). For examples, see [Build resilient HTTP apps: Key development patterns](#).

Metrics enrichment

Enrichment is the automatic augmentation of telemetry with well-known state, in the form of name/value pairs. For example, an app might emit a log that includes the *operation* and *result code* as columns to represent the outcome of some operation. In this situation and depending on peripheral context, enrichment adds *Cluster name*, *Process name*, *Region*, *Tenant ID*, and more to the log as it's sent to the telemetry backend. When enrichment is added, the app code doesn't need to do anything extra to benefit from enriched metrics.

How enrichment works

Imagine 1,000 globally distributed service instances generating logs and metrics. When you encounter an issue on your [service dashboard](#), it's crucial to quickly identify the problematic region or data center. Enrichment ensures that metric records contain the necessary information to pinpoint failures in distributed systems. Without enrichment, the burden falls on the app code to internally manage this state, integrate it into the logging process, and manually transmit it. Enrichment simplifies this process, seamlessly handling it without affecting the app's logic.

In the case of resiliency, when you add enrichment the following dimensions are added to the outgoing telemetry:

- `error.type`: Low-cardinality version of an exception's information.
- `request.name`: The name of the request.
- `request.dependency.name`: The name of the dependency.

Under the covers, resilience enrichment is built on top of Polly's Telemetry MeteringEnricher. For more information, see [Polly: Metering enrichment ↗](#).

Add resilience enrichment

In addition to registering a resilience pipeline, you can also register resilience enrichment. To add enrichment, call the [AddResilienceEnricher\(IServiceCollection\)](#) extensions method on the `IServiceCollection` instance.

```
C#
```

```
services.AddResilienceEnricher();
```

By calling the `AddResilienceEnricher` extension method, you're adding dimensions on top of the default ones that are built into the underlying Polly library. The following enrichment dimensions are added:

- Exception enrichment based on the [IExceptionSummarizer](#), which provides a mechanism to summarize exceptions for use in telemetry. For more information, see [Exception summarization](#).
- Request metadata enrichment based on [RequestMetadata](#), which holds the request metadata for telemetry. For more information, see [Polly: Telemetry metrics ↗](#).

Use resilience pipeline

To use a configured resilience pipeline, you must get the pipeline from a `ResiliencePipelineProvider< TKey >`. When you added the pipeline earlier, the `key` was of type `string`, so you must get the pipeline from the `ResiliencePipelineProvider< string >`.

C#

```
using ServiceProvider provider = services.BuildServiceProvider();

ResiliencePipelineProvider<string> pipelineProvider =
    provider.GetRequiredService<ResiliencePipelineProvider<string>>();

ResiliencePipeline pipeline = pipelineProvider.GetPipeline(key);
```

The preceding code:

- Builds a `ServiceProvider` from the `ServiceCollection` instance.
- Gets the `ResiliencePipelineProvider< string >` from the service provider.
- Retrieves the `ResiliencePipeline` from the `ResiliencePipelineProvider< string >`.

Execute resilience pipeline

To use the resilience pipeline, call any of the available `Execute*` methods on the `ResiliencePipeline` instance. For example, consider an example call to `ExecuteAsync` method:

C#

```
await pipeline.ExecuteAsync(static cancellationToken =>
{
    // Code that could potentially fail.

    return ValueTask.CompletedTask;
});
```

The preceding code executes the delegate within the `ExecuteAsync` method. When there are failures, the configured strategies are executed. For example, if the `RetryStrategy` is configured to retry three times, the delegate is executed four times (one initial attempt plus three retry attempts) before the failure is propagated.

Next steps

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

.NET is an open source project.
Select a link to provide feedback:

-  [Open a documentation issue](#)
-  [Provide product feedback](#)

Network programming in .NET

Article • 10/27/2022

.NET provides a layered, extensible, and managed implementation of Internet services that can be quickly and easily integrated into your apps. Your network apps can build on pluggable protocols to automatically take advantage of various Internet protocols, or they can use a managed implementation of cross-platform socket interfaces to work with the network on the socket level.

Internet apps

Internet apps can be classified broadly into two kinds: client apps that request information and server apps that respond to information requests from clients. The classic Internet client-server app is the World Wide Web, where people use browsers to access documents and other data stored on web servers worldwide.

Apps are not limited to just one of these roles; for instance, the familiar middle-tier app server responds to requests from clients by requesting data from another server, in which case it is acting as both a server and a client.

The client app requests by identifying the requested Internet resource and the communication protocol to use for the request and response. If necessary, the client also provides any additional data required to complete the request, such as proxy location or authentication information (user name, password, and so on). Once the request is formed, the request can be sent to the server.

Identifying resources

.NET uses a uniform resource identifier (URI) to identify the requested Internet resource and communication protocol. The URI consists of at least three, and possibly four, fragments: the scheme identifier, which identifies the communications protocol for the request and response; the server identifier, which consists of either a domain name system (DNS) hostname or a TCP address that uniquely identifies the server on the Internet; the path identifier, which locates the requested information on the server; and an optional query string, which passes information from the client to the server.

The [System.Uri](#) type is used as a representation of a uniform resource identifier (URI) and easy access to the parts of the URI. To create a `Uri` instance you can pass it a string:

C#

```
const string uriString =
    "https://learn.microsoft.com/en-us/dotnet/path?key=value#bookmark";

Uri canonicalUri = new(uriString);
Console.WriteLine(canonicalUri.Host);
Console.WriteLine(canonicalUri.PathAndQuery);
Console.WriteLine(canonicalUri.Fragment);
// Sample output:
//      learn.microsoft.com
//      /en-us/dotnet/path?key=value
//      #bookmark
```

The `Uri` class automatically performs validation and canonicalization per [RFC 3986](#). These validation and canonicalization rules are used to ensure that a URI is well-formed and that the URI is in a canonical form.

See also

- [Runtime configuration options for networking](#)
- [HTTP support in .NET](#)
- [Sockets in .NET](#)
- [TCP in .NET](#)
- [Tutorial: Make HTTP requests in a .NET console app using C#](#)
- [Networking telemetry in .NET](#)
- [.NET Networking improvements](#)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

.NET is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

Network availability

Article • 08/27/2022

The [System.Net.NetworkInformation](#) namespace enables you to gather information about network events, changes, statistics, and properties. In this article, you'll learn how to use the [System.Net.NetworkInformation.NetworkChange](#) class to determine whether the network address or availability has changed. Additionally, you'll see about the network statistics and properties on an interface or protocol basis. Finally, you'll use the [System.Net.NetworkInformation.Ping](#) class to determine whether a remote host is reachable.

Network change events

The [System.Net.NetworkInformation.NetworkChange](#) class enables you to determine whether the network address or availability has changed. To use this class, create an event handler to process the change, and associate it with a [NetworkAddressChangedEventHandler](#) or a [NetworkAvailabilityChangedEventHandler](#).

C#

```
NetworkChange.NetworkAvailabilityChanged += OnNetworkAvailabilityChanged;

static void OnNetworkAvailabilityChanged(
    object? sender, NetworkAvailabilityEventArgs networkAvailability) =>
    Console.WriteLine($"Network is available:
{networkAvailability.IsAvailable}");

Console.WriteLine(
    "Listening changes in network availability. Press any key to
continue.");
Console.ReadLine();

NetworkChange.NetworkAvailabilityChanged -= OnNetworkAvailabilityChanged;
```

The preceding C# code:

- Registers an event handler for the [NetworkChange.NetworkAvailabilityChanged](#) event.
- The event handler simply writes the availability status to the console.
- A message is written to the console letting the user know that the code is listening for changes in network availability and waits for a key press to exit.
- Unregisters the event handler.

C#

```
NetworkChange.NetworkAddressChanged += OnNetworkAddressChanged;

static void OnNetworkAddressChanged(
    object? sender, EventArgs args)
{
    foreach ((string name, OperationalStatus status) in
        NetworkInterface.GetAllNetworkInterfaces()
            .Select(networkInterface =>
                (networkInterface.Name,
                networkInterface.OperationalStatus)))
    {
        Console.WriteLine(
            $"{name} is {status}");
    }
}

Console.WriteLine(
    "Listening for address changes. Press any key to continue.");
Console.ReadLine();

NetworkChange.NetworkAddressChanged -= OnNetworkAddressChanged;
```

The preceding C# code:

- Registers an event handler for the `NetworkChange.NetworkAddressChanged` event.
- The event handler iterates over `NetworkInterface.GetAllNetworkInterfaces()`, writing their name and operational status to the console.
- A message is written to the console letting the user know that the code is listening for changes in network availability and waits for a key press to exit.
- Unregisters the event handler.

Network statistics and properties

You can gather network statistics and properties on an interface or protocol basis. The `NetworkInterface`, `NetworkInterfaceType`, and `PhysicalAddress` classes give information about a particular network interface, while the `IPInterfaceProperties`, `IPGlobalProperties`, `IPGlobalStatistics`, `TcpStatistics`, and `UdpStatistics` classes give information about layer 3 and layer 4 packets.

C#

```
ShowStatistics(NetworkInterfaceComponent.IPv4);
ShowStatistics(NetworkInterfaceComponent.IPv6);
```

```

static void ShowStatistics(NetworkInterfaceComponent version)
{
    var properties = IPGlobalProperties.GetIPGlobalProperties();
    var stats = version switch
    {
        NetworkInterfaceComponent.IPv4 => properties.GetTcpIPv4Statistics(),
        _ => properties.GetTcpIPv6Statistics()
    };

    Console.WriteLine($"TCP/{version} Statistics");
    Console.WriteLine($"  Minimum Transmission Timeout : {stats.MinimumTransmissionTimeout:#,#}");
    Console.WriteLine($"  Maximum Transmission Timeout : {stats.MaximumTransmissionTimeout:#,#}");
    Console.WriteLine("  Connection Data");
    Console.WriteLine($"    Current : {stats.CurrentConnections:#,#}");
    Console.WriteLine($"    Cumulative : {stats.CumulativeConnections:#,#}");
    Console.WriteLine($"    Initiated : {stats.ConnectionsInitiated:#,#}");
    Console.WriteLine($"    Accepted : {stats.ConnectionsAccepted:#,#}");
    Console.WriteLine($"    Failed Attempts : {stats.FailedConnectionAttempts:#,#}");
    Console.WriteLine($"    Reset : {stats.ResetConnections:#,#}");
    Console.WriteLine("  Segment Data");
    Console.WriteLine($"    Received : {stats.SegmentsReceived:#,#}");
    Console.WriteLine($"    Sent : {stats.SegmentsSent:#,#}");
    Console.WriteLine($"    Retransmitted : {stats.SegmentsResent:#,#}");
    Console.WriteLine();
}

```

The preceding C# code:

- Calls a custom `ShowStatistics` method to display the statistics for each protocol.
- The `ShowStatistics` method calls `IPGlobalProperties.GetIPGlobalProperties()`, and depending on the given `NetworkInterfaceComponent` will either call `IPGlobalProperties.GetIPv4GlobalStatistics()` or `IPGlobalProperties.GetIPv6GlobalStatistics()`.
- The `TcpStatistics` are written to the console.

Determine if a remote host is reachable

You can use the [Ping](#) class to determine whether a remote host is up, on the network, and reachable.

C#

```
using Ping ping = new();

string hostName = "stackoverflow.com";
PingReply reply = await ping.SendPingAsync(hostName);
Console.WriteLine($"Ping status for ({hostName}): {reply.Status}");
if (reply is { Status: IPStatus.Success })
{
    Console.WriteLine($"Address: {reply.Address}");
    Console.WriteLine($"Roundtrip time: {reply.RoundtripTime}");
    Console.WriteLine($"Time to live: {reply.Options?.Ttl}");
    Console.WriteLine();
}
```

The preceding C# code:

- Instantiate a [Ping](#) object.
- Calls [Ping.SendPingAsync\(String\)](#) with the `"stackoverflow.com"` hostname parameter.
- The status of the ping is written to the console.

See also

- [Network programming in .NET](#)
- [NetworkInterface](#)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

Open a documentation issue

Provide product feedback

Internet Protocol version 6 (IPv6) overview

Article • 10/07/2022

The Internet Protocol version 6 (IPv6) is a suite of standard protocols for the network layer of the Internet. IPv6 is designed to solve many of the problems of the current version of the Internet Protocol suite (known as IPv4) about address depletion, security, auto-configuration, extensibility, and so on. IPv6 expands the capabilities of the Internet to enable new kinds of applications, including peer-to-peer and mobile applications.

The following are the main issues of the current IPv4 protocol:

- Rapid depletion of the address space.

This has led to the use of Network Address Translators (NATs) that map multiple private addresses to a single public IP address. The main problems created by this mechanism are processing overhead and lack of end-to-end connectivity.

- Lack of hierarchy support.

Because of its inherent predefined class organization, IPv4 lacks true hierarchical support. It is impossible to structure the IP addresses in a way that truly maps the network topology. This crucial design flaw creates the need for large routing tables to deliver IPv4 packets to any location on the Internet.

- Complex network configuration.

With IPv4, addresses must be assigned statically or using a configuration protocol such as DHCP. In an ideal situation, hosts would not have to rely on the administration of a DHCP infrastructure. Instead, they would be able to configure themselves based on the network segment in which they are located.

- Lack of built-in authentication and confidentiality.

IPv4 does not require support for any mechanism that provides authentication or encryption of the exchanged data. This changes with IPv6. Internet Protocol security (IPSec) is an IPv6 support requirement.

A new protocol suite must satisfy the following basic requirements:

- Large-scale routing and addressing with low overhead.
- Auto-configuration for various connecting situations.
- Built-in authentication and confidentiality.

IPv6 addressing

With IPv6, addresses are 128 bits long. One reason for such a large address space is to subdivide the available addresses into a hierarchy of routing domains that reflect the Internet's topology. Another reason is to map the addresses of network adapters (or interfaces) that connect devices to the network. IPv6 features an inherent capability to resolve addresses at their lowest level, which is at the network interface level, and also has auto-configuration capabilities.

Text representation

The following are the three conventional forms used to represent the IPv6 addresses as text strings:

- **Colon-hexadecimal form:**

This is the preferred form `n:n:n:n:n:n:n:n`. Each `n` represents the hexadecimal value of one of the eight 16-bit elements of the address. For example:

`3FFE:FFFF:7654:FEDA:1245:BA98:3210:4562`.

- **Compressed form:**

Due to the address length, it is common to have addresses containing a long string of zeros. To simplify writing these addresses, use the compressed form, in which a single contiguous sequence of 0 blocks is represented by a double-colon symbol (`::`). This symbol can appear only once in an address. For example, the multicast address `FFED:0:0:0:0:BA98:3210:4562` in compressed form is

`FFED::BA98:3210:4562`. The unicast address `3FFE:FFFF:0:0:8:800:20C4:0` in compressed form is `3FFE:FFFF::8:800:20C4:0`. The loopback address `0:0:0:0:0:0:1` in compressed form is `::1`. The unspecified address `0:0:0:0:0:0:0:0` in compressed form is `::`.

- **Mixed form:**

This form combines IPv4 and IPv6 addresses. In this case, the address format is `n:n:n:n:n:n:d.d.d.d`, where each `n` represents the hexadecimal values of the six IPv6 high-order 16-bit address elements, and each `d` represents the decimal value of an IPv4 address.

Address types

The leading bits in the address define the specific IPv6 address type. The variable-length field containing these leading bits is called a Format Prefix (FP).

An IPv6 unicast address is divided into two parts. The first part contains the address prefix, and the second part contains the interface identifier. A concise way to express an IPv6 address/prefix combination is as follows: ipv6-address/prefix-length.

The following is an example of an address with a 64-bit prefix.

`3FFE:FFFF:0:CD30:0:0:0:0/64`.

The prefix in this example is `3FFE:FFFF:0:CD30`. The address can also be written in a compressed form, as `3FFE:FFFF:0:CD30::/64`.

IPv6 defines the following address types:

- **Unicast address:**

An identifier for a single interface. A packet sent to this address is delivered to the identified interface. The unicast addresses are distinguished from the multicast addresses by the value of the high-order octet. The multicast addresses' high-order octet has the hexadecimal value of FF. Any other value for this octet identifies a unicast address. The following are different types of unicast addresses:

- **Link-local addresses:**

These addresses are used on a single link and have the following format:

`FE80::*InterfaceID*`. Link-local addresses are used between nodes on a link for auto-address configuration, neighbor discovery, or when no routers are present. A link-local address is used primarily at startup and when the system has not yet acquired addresses of larger scope.

- **Site-local addresses:**

These addresses are used on a single site and have the following format:

`FEC0::*SubnetID*::*InterfaceID*`. The site-local addresses are used for addressing inside a site without the need for a global prefix.

- **Global IPv6 unicast addresses:**

These addresses can be used across the Internet and have the following format:

`*GlobalRoutingPrefix*::*SubnetID*::*InterfaceID*`.

- **Multicast address:**

An identifier for a set of interfaces (typically belonging to different nodes). A packet sent to this address is delivered to all the interfaces identified by the address. The multicast address types supersede the IPv4 broadcast addresses.

- **Anycast address:**

An identifier for a set of interfaces (typically belonging to different nodes). A packet sent to this address is delivered to only one interface identified by the address. This is the nearest interface as identified by routing metrics. Anycast addresses are taken from the unicast address space and are not syntactically distinguishable. The addressed interface performs the distinction between unicast and anycast addresses as a function of its configuration.

In general, a node always has a link-local address. It might have a site-local address and one or more global addresses.

IPv6 routing

A flexible routing mechanism is a benefit of IPv6. Due to how IPv4 network IDs were and are allocated, large routing tables need to be maintained by the routers that are on the Internet backbones. These routers must know all the routes to forward packets that are potentially directed to any node on the Internet. With its ability to aggregate addresses, IPv6 allows flexible addressing and drastically reduces the size of routing tables. In this new addressing architecture, intermediate routers must keep track only of the local portion of their network to forward the messages appropriately.

Neighbor discovery

Some of the features provided by *neighbor discovery* are:

- **Router discovery:** This allows hosts to identify local routers.
- **Address resolution:** This allows nodes to resolve a link-layer address for a corresponding next-hop address (a replacement for Address Resolution Protocol [ARP]).
- **Address auto-configuration:** This allows hosts to automatically configure site-local and global addresses.

Neighbor discovery uses Internet Control Message Protocol for IPv6 (ICMPv6) messages that include:

- **Router advertisement:** Sent by a router on a pseudo-periodic basis or in response to a router solicitation. IPv6 routers use router advertisements to advertise their

availability, address prefixes, and other parameters.

- **Router solicitation:** Sent by a host to request that routers on the link send a router advertisement immediately.
- **Neighbor solicitation:** Sent by nodes for address resolution, duplicate address detection, or to verify that a neighbor is still reachable.
- **Neighbor advertisement:** Sent by nodes to respond to a neighbor solicitation or to notify neighbors of a change in link-layer address.
- **Redirect:** Sent by routers to indicate a better next-hop address to a particular destination for a sending node.

IPv6 auto-configuration

One important goal for IPv6 is to support node Plug and Play. That is, it should be possible to plug a node into an IPv6 network and have it automatically configured without any human intervention.

Auto-configuration types

IPv6 supports the following types of auto-configuration:

- **Stateful auto-configuration:**

This type of configuration requires a certain level of human intervention because it needs a Dynamic Host Configuration Protocol for IPv6 (DHCPv6) server for the installation and administration of the nodes. The DHCPv6 server keeps a list of nodes to which it supplies configuration information. It also maintains state information so the server knows how long each address is in use, and when it might be available for reassignment.

- **Stateless auto-configuration:**

This type of configuration is suitable for small organizations and individuals. In this case, each host determines its addresses from the contents of received router advertisements. Using the IEEE EUI-64 standard to define the network ID portion of the address, it is reasonable to assume the uniqueness of the host address on the link.

Regardless of how the address is determined, the node must verify that its potential address is unique to the local link. This is done by sending a neighbor solicitation message to the potential address. If the node receives any response, it knows that the address is already in use and must determine another address.

IPv6 mobility

The proliferation of mobile devices has introduced a new requirement: A device must be able to arbitrarily change locations on the IPv6 Internet and still maintain existing connections. To provide this functionality, a mobile node is assigned a home address at which it can always be reached. When the mobile node is at home, it connects to the home link and uses its home address. When the mobile node is away from home, a home agent, which is usually a router, relays messages between the mobile node and the nodes with which it is communicating.

Disable or enable IPv6

To use the IPv6 protocol, ensure that you are running a version of the operating system that supports IPv6 and ensure that the operating system and the networking classes are configured properly.

Configuration Steps

The following table lists various configurations

| OS IPv6 enabled? | Code IPv6 enabled? | Description |
|------------------|--------------------|--|
| ✗ No | ✗ No | Can parse IPv6 addresses. |
| ✗ No | ✓ Yes | Can parse IPv6 addresses. |
| ✓ Yes | ✗ No | Can parse IPv6 addresses and resolve IPv6 addresses using name resolution methods not marked obsolete. |
| ✓ Yes | ✓ Yes | Can parse and resolve IPv6 addresses using all methods including those marked obsolete. |

IPv6 is enabled by default. To configure this switch in an environment variable, use the `DOTNET_SYSTEM_NET_DISABLEIPV6` environment variable. For more information, see [.NET environment variables: DOTNET_SYSTEM_NET_DISABLEIPV6](#).

See also

- [Networking in .NET](#)
- [Sockets in .NET](#)
- [System.AppContext](#)

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

Service discovery in .NET

Article • 12/11/2023

In this article, you learn how to use the `Microsoft.Extensions.ServiceDiscovery` library. Service discovery is a way for developers to use logical names instead of physical addresses (IP address and port) to refer to external services.

Get Started

To get started with service discovery in .NET, install the [Microsoft.Extensions.ServiceDiscovery](#) NuGet package.

.NET CLI

.NET CLI

```
dotnet add package Microsoft.Extensions.ServiceDiscovery --prerelease
```

For more information, see [dotnet add package](#) or [Manage package dependencies in .NET applications](#).

Example usage

In the `Program.cs` file of your project, call the [AddServiceDiscovery](#) extension method to add service discovery to the host, configuring default service endpoint resolvers:

C#

```
builder.Services.AddServiceDiscovery();
```

Add service discovery to an individual [IHttpClientBuilder](#) by calling the [UseServiceDiscovery](#) extension method:

C#

```
builder.Services.AddHttpClient<CatalogServiceClient>(static client =>
{
    client.BaseAddress = new("http://catalog");
})
.UseServiceDiscovery();
```

Alternatively, you can add service discovery to all [HttpClient](#) instances by default:

C#

```
builder.Services.ConfigureHttpClientDefaults(static http =>
{
    // Turn on service discovery by default
    http.UseServiceDiscovery();
});
```

Resolve service endpoints from configuration

The [AddServiceDiscovery](#) extension method adds a configuration-based endpoint resolver by default. This resolver reads endpoints from the [.NET configuration system](#). The library supports configuration through *appsettings.json*, environment variables, or any other [IConfiguration](#) source.

Here's an example demonstrating how to configure endpoints for the service named *catalog* via [appsettings.json](#):

JSON

```
{
  "Services": {
    "catalog": [
      "localhost:8080",
      "10.46.24.90:80",
    ]
  }
}
```

The preceding example adds two endpoints for the service named *catalog*: `localhost:8080`, and `"10.46.24.90:80"`. Each time the *catalog* is resolved, one of these endpoints is selected.

If service discovery was added to the host using the [AddServiceDiscoveryCore](#) extension method on [IServiceCollection](#), the configuration-based endpoint resolver can be added by calling the [AddConfigurationServiceEndPointResolver](#) extension method on [IServiceCollection](#).

Configuration

The configuration resolver is configured using the [ConfigurationServiceEndPointOptionsResolver](#) class, which offers these configuration

options:

- **SectionName**: The name of the configuration section that contains service endpoints. It defaults to "Services".
- **ApplyHostNameMetadata**: A delegate used to determine if host name metadata should be applied to resolved endpoints. It defaults to a function that returns `false`.

To configure these options, you can use the `Configure` extension method on the `IServiceCollection` within your application's `Startup` class or `Program` file:

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.Configure<ConfigurationServiceEndPointResolverOptions>(
    static options =>
{
    options.SectionName = "MyServiceEndpoints";

    // Configure the logic for applying host name metadata
    options.ApplyHostNameMetadata = static endpoint =>
    {
        // Your custom logic here. For example:
        return endpoint.EndPoint is DnsEndPoint dnsEp
            && dnsEp.Host.StartsWith("internal");
    };
});
```

The preceding example demonstrates setting a custom section name for your service endpoints and providing custom conditional logic for applying host name metadata.

Resolve service endpoints using platform-provided service discovery

Certain platforms, like Azure Container Apps and Kubernetes (when configured accordingly), offer service discovery capabilities without necessitating a service discovery client library. In cases where an application is deployed in such environments, using the platform's built-in functionality can be advantageous. The pass-through resolver is designed to facilitate this scenario. It enables the utilization of alternative resolvers, such as configuration, in different environments, such as a developer's machine. Importantly, this flexibility is achieved without the need for any code modifications or the implementation of conditional guards.

The pass-through resolver performs no external resolution and instead resolves endpoints by returning the input service name represented as a [DnsEndPoint](#).

The pass-through provider is configured by-default when adding service discovery via the `AddServiceDiscovery` extension method.

If service discovery was added to the host using the `AddServiceDiscoveryCore` extension method on `IServiceCollection`, the pass-through provider can be added by calling the `AddPassThroughServiceEndPointResolver` extension method on `IServiceCollection`.

Load-balancing with endpoint selectors

Each time an endpoint is resolved via the `HttpClient` pipeline, a single endpoint is selected from the set of all known endpoints for the requested service. If multiple endpoints are available, it may be desirable to balance traffic across all such endpoints. To accomplish this, a customizable *endpoint selector* can be used. By default, endpoints are selected in round-robin order. To use a different endpoint selector, provide an `IServiceEndPointSelector` instance to the [UseServiceDiscovery](#) method call. For example, to select a random endpoint from the set of resolved endpoints, specify `RandomServiceEndPointSelectorProvider.Instance` as the endpoint selector:

```
C#
```

```
builder.Services.AddHttpClient<CatalogServiceClient>(
    static client => client.BaseAddress = new("http://catalog")
)
.UseServiceDiscovery(RandomServiceEndPointSelectorProvider.Instance);
```

The `Microsoft.Extensions.ServiceDiscovery` package includes the following endpoint selector providers:

- `PickFirstServiceEndPointSelectorProvider.Instance`: Pick-first, which always selects the first endpoint.
- `RoundRobinServiceEndPointSelectorProvider.Instance`: Round-robin, which cycles through endpoints.
- `RandomServiceEndPointSelectorProvider.Instance`: Random, which selects endpoints randomly.
- `PowerOfTwoChoicesServiceEndPointSelectorProvider.Instance`: Power-of-two-choices, which attempt to pick the least used endpoint based on the *Power of Two Choices* algorithm for distributed load balancing, degrading to randomly selecting an endpoint when either of the provided endpoints don't have the `IEndPointLoadFeature` feature.

Endpoint selectors are created via an `IServiceEndPointSelectorProvider` instance, such as the providers previously listed. The provider's `CreateSelector()` method is called to create a selector, which is an instance of `IServiceEndPointSelector`. The `IServiceEndPointSelector` instance is given the set of known endpoints when they're resolved, using the `IServiceEndPointSelector.SetEndPoints` method. To choose an endpoint from the collection, the `IServiceEndPointSelector.GetEndPoint` method is called, returning a single `ServiceEndPoint`. The `context` value passed to `GetEndPoint` is used to provide extra context that may be useful to the selector. For example, in the `HttpClient` case, the `HttpRequestMessage` is passed. None of the provided implementations of `IServiceEndPointSelector` inspect the context, and it can be ignored unless you're using a selector, which does make use of it.

Resolution order

When service endpoints are being resolved, each registered resolver is called in the order of registration and given the opportunity to modify the collection of `ServiceEndPoints` which are returned back to the caller. The providers included in the `Microsoft.Extensions.ServiceDiscovery` series of packages skip resolution if there are existing endpoints in the collection when they're called. For example, consider a case where the following providers are registered: *Configuration, DNS SRV, Pass-through*. When resolution occurs, the providers are called in-order. If the *Configuration* providers discover no endpoints, the *DNS SRV* provider performs resolution and may add one or more endpoints. If the *DNS SRV* provider adds an endpoint to the collection, the *Pass-through* provider skips its resolution and returns immediately instead.

See also

- [Service discovery in .NET Aspire](#)

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

.NET is an open source project.
Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

HTTP support in .NET

Article • 05/22/2023

Hypertext Transfer Protocol (or HTTP) is a protocol for requesting resources from a web server. The [System.Net.Http.HttpClient](#) class exposes the ability to send HTTP requests and receive HTTP responses from a resource identified by a URI. Many types of resources are available on the web, and HTTP defines a set of request methods for accessing these resources.

HTTP request methods

The request methods are differentiated via several factors, first by their *verb* but also by the following characteristics:

- A request method is *idempotent* if it can be successfully processed multiple times without changing the result. For more information, see [RFC 9110: 9.2.2. Idempotent Methods](#).
- A request method is *cacheable* when its corresponding response can be stored for reuse. For more information, see [RFC 9110: Section 9.2.3. Methods and Caching](#).
- A request method is considered a *safe method* if it doesn't modify the state of a resource. All *safe methods* are also *idempotent*, but not all *idempotent* methods are considered *safe*. For more information, see [RFC 9110: Section 9.2.1. Safe Methods](#).

| HTTP method | Is idempotent | Is cacheable | Is safe |
|-------------|---------------|-----------------------|---------|
| GET | ✓ Yes | ✓ Yes | ✓ Yes |
| POST | ✗ No | ⚠ [†] Rarely | ✗ No |
| PUT | ✓ Yes | ✗ No | ✗ No |
| PATCH | ✗ No | ✗ No | ✗ No |
| DELETE | ✓ Yes | ✗ No | ✗ No |
| HEAD | ✓ Yes | ✓ Yes | ✓ Yes |
| OPTIONS | ✓ Yes | ✗ No | ✓ Yes |
| TRACE | ✓ Yes | ✗ No | ✓ Yes |
| CONNECT | ✗ No | ✗ No | ✗ No |

[†]The `POST` method is only cacheable when the appropriate `Cache-Control` or `Expires` response headers are present. This is very uncommon in practice.

HTTP status codes

.NET provides comprehensive support for the HTTP protocol, which accounts for most internet traffic, with the `HttpClient`. For more information, see [Make HTTP requests with the HttpClient class](#). Applications receive HTTP protocol errors by catching an `HttpRequestException`. HTTP status codes are either reported in `HttpResponseMessage` with the `HttpResponseMessage.StatusCode` or in `HttpRequestException` with the `HttpRequestException.StatusCode` in case the called method doesn't return a response message. For more information about error handling, see [HTTP error handling](#), and for more information about status codes, see [RFC 9110, HTTP Semantics: Status Codes](#).

Informational status codes

The informational status codes reflect an interim response. Most of the interim responses, for example `StatusCode.Continue`, are handled internally with `HttpClient` and are never surfaced to the user.

| HTTP status code | <code>HttpStatusCode</code> |
|------------------|--|
| 100 | <code>(HttpStatusCode.Continue)</code> |
| 101 | <code>(HttpStatusCode.SwitchingProtocols)</code> |
| 102 | <code>(HttpStatusCode.Processing)</code> |
| 103 | <code>(HttpStatusCode.EarlyHints)</code> |

Successful status codes

The successful status codes indicate that the client's request was successfully received, understood, and accepted.

| HTTP status code | <code>HttpStatusCode</code> |
|------------------|---|
| 200 | <code>(HttpStatusCode.OK)</code> |
| 201 | <code>(HttpStatusCode.Created)</code> |
| 202 | <code>(HttpStatusCode.Accepted)</code> |
| 203 | <code>(HttpStatusCode.NonAuthoritativeInformation)</code> |

| | |
|-------------------------|--------------------------------|
| HTTP status code | HttpStatusCode |
| 204 | HttpStatusCode.NoContent |
| 205 | HttpStatusCode.ResetContent |
| 206 | HttpStatusCode.PartialContent |
| 207 | HttpStatusCode.MultiStatus |
| 208 | HttpStatusCode.AlreadyReported |
| 226 | HttpStatusCode.IMUsed |

Redirection status codes

Redirection status codes require the user agent to take action to fulfill the request. Automatic redirection is turned on by default, it can be changed with [HttpClientHandler.AllowAutoRedirect](#) or [SocketsHttpHandler.AllowAutoRedirect](#).

| | |
|-------------------------|---|
| HTTP status code | HttpStatusCode |
| 300 | HttpStatusCode.MultipleChoices or HttpStatusCode.Ambiguous |
| 301 | HttpStatusCode.MovedPermanently or HttpStatusCode.Moved |
| 302 | HttpStatusCode.Found or HttpStatusCode.Redirect |
| 303 | HttpStatusCode.SeeOther or HttpStatusCode.RedirectMethod |
| 304 | HttpStatusCode.NotModified |
| 305 | HttpStatusCode.UseProxy |
| 306 | HttpStatusCode.Unused |
| 307 | HttpStatusCode.TemporaryRedirect or HttpStatusCode.RedirectKeepVerb |
| 308 | HttpStatusCode.PermanentRedirect |

Client error status codes

The client error status codes indicate that the client's request was invalid.

| | |
|-------------------------|-----------------------------|
| HTTP status code | HttpStatusCode |
| 400 | HttpStatusCode.BadRequest |
| 401 | HttpStatusCode.Unauthorized |

| HTTP status code | HttpStatusCode |
|------------------|---|
| 402 | HttpStatusCode.PaymentRequired |
| 403 | HttpStatusCode.Forbidden |
| 404 | HttpStatusCode.NotFound |
| 405 | HttpStatusCode.MethodNotAllowed |
| 406 | HttpStatusCode.NotAcceptable |
| 407 | HttpStatusCode.ProxyAuthenticationRequired |
| 408 | HttpStatusCode.RequestTimeout |
| 409 | HttpStatusCode.Conflict |
| 410 | HttpStatusCode.Gone |
| 411 | HttpStatusCode.LengthRequired |
| 412 | HttpStatusCode.PreconditionFailed |
| 413 | HttpStatusCode.RequestEntityTooLarge |
| 414 | HttpStatusCode.RequestUriTooLong |
| 415 | HttpStatusCode.UnsupportedMediaType |
| 416 | HttpStatusCode.RequestedRangeNotSatisfiable |
| 417 | HttpStatusCode.ExpectationFailed |
| 418 | I'm a teapot ↴ □ |
| 421 | HttpStatusCode.MisdirectedRequest |
| 422 | HttpStatusCode.UnprocessableEntity |
| 423 | HttpStatusCode.Locked |
| 424 | HttpStatusCode.FailedDependency |
| 426 | HttpStatusCode.UpgradeRequired |
| 428 | HttpStatusCode.PreconditionRequired |
| 429 | HttpStatusCode.TooManyRequests |
| 431 | HttpStatusCode.RequestHeaderFieldsTooLarge |
| 451 | HttpStatusCode.UnavailableForLegalReasons |

HTTP status code

HttpStatusCode

Server error status codes

The server error status codes indicate that the server encountered an unexpected condition that prevented it from fulfilling the request.

| HTTP status code | HttpStatusCode |
|------------------|--|
| 500 | HttpStatusCode.InternalServerError |
| 501 | HttpStatusCode.NotImplemented |
| 502 | HttpStatusCode.BadGateway |
| 503 | HttpStatusCode.ServiceUnavailable |
| 504 | HttpStatusCode.GatewayTimeout |
| 505 | HttpStatusCode.HttpVersionNotSupported |
| 506 | HttpStatusCode.VariantAlsoNegotiates |
| 507 | HttpStatusCode.InsufficientStorage |
| 508 | HttpStatusCode.LoopDetected |
| 510 | HttpStatusCode.NotExtended |
| 511 | HttpStatusCode.NetworkAuthenticationRequired |

See also

- [Make HTTP requests with the HttpClient class](#)
- [HTTP client factory with .NET](#)
- [Guidelines for using HttpClient](#)
- [.NET Networking improvements ↗](#)

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

more information, see [our contributor guide](#).

 [Provide product feedback](#)

Guidelines for using HttpClient

Article • 09/01/2023

The [System.Net.Http.HttpClient](#) class sends HTTP requests and receives HTTP responses from a resource identified by a URI. An [HttpClient](#) instance is a collection of settings that's applied to all requests executed by that instance, and each instance uses its own connection pool, which isolates its requests from others. Starting in .NET Core 2.1, the [SocketsHttpHandler](#) class provides the implementation, making behavior consistent across all platforms.

DNS behavior

[HttpClient](#) only resolves DNS entries when a connection is created. It does not track any time to live (TTL) durations specified by the DNS server. If DNS entries change regularly, which can happen in some scenarios, the client won't respect those updates. To solve this issue, you can limit the lifetime of the connection by setting the [PooledConnectionLifetime](#) property, so that DNS lookup is repeated when the connection is replaced. Consider the following example:

C#

```
var handler = new SocketsHttpHandler
{
    PooledConnectionLifetime = TimeSpan.FromMinutes(15) // Recreate every 15
    minutes
};
var sharedClient = new HttpClient(handler);
```

The preceding `HttpClient` is configured to reuse connections for 15 minutes. After the timespan specified by [PooledConnectionLifetime](#) has elapsed, the connection is closed and a new one is created.

Pooled connections

The connection pool for an [HttpClient](#) is linked to the underlying [SocketsHttpHandler](#). When the [HttpClient](#) instance is disposed, it disposes all existing connections inside the pool. If you later send a request to the same server, a new connection must be recreated. As a result, there's a performance penalty for unnecessary connection creation. Moreover, TCP ports are not released immediately after connection closure. (For more information on that, see TCP `TIME-WAIT` in [RFC 9293](#).) If the rate of requests

is high, the operating system limit of available ports might be exhausted. To avoid port exhaustion problems, we [recommend](#) reusing `HttpClient` instances for as many HTTP requests as possible.

Recommended use

To summarize recommended `HttpClient` use in terms of lifetime management, you should use either *long-lived* clients and set `PooledConnectionLifetime` (.NET Core and .NET 5+) or *short-lived* clients created by `IHttpClientFactory`.

- In .NET Core and .NET 5+:
 - Use a `static` or *singleton* `HttpClient` instance with `PooledConnectionLifetime` set to the desired interval, such as 2 minutes, depending on expected DNS changes. This solves both the port exhaustion and DNS changes problems without adding the overhead of `IHttpClientFactory`. If you need to be able to mock your handler, you can register it separately.

💡 Tip

If you only use a limited number of `HttpClient` instances, that's also an acceptable strategy. What matters is that they're not created and disposed with each request, as they each contain a connection pool. Using more than one instance is necessary for scenarios with multiple proxies or to separate cookie containers without completely disabling cookie handling.

- Using `IHttpClientFactory`, you can have multiple, differently configured clients for different use cases. However, be aware that the factory-created clients are intended to be short-lived, and once the client is created, the factory no longer has control over it.

The factory pools `HttpMessageHandler` instances, and, if its lifetime hasn't expired, a handler can be reused from the pool when the factory creates a new `HttpClient` instance. This reuse avoids any socket exhaustion issues.

If you desire the configurability that `IHttpClientFactory` provides, we recommend using the [typed-client approach](#).

- In .NET Framework, use `IHttpClientFactory` to manage your `HttpClient` instances. If you don't use the factory and instead create a new client instance for each request yourself, you can exhaust available ports.

💡 Tip

If your app requires cookies, consider disabling automatic cookie handling or avoiding `IHttpClientFactory`. Pooling the `HttpMessageHandler` instances results in sharing of `CookieContainer` objects. Unanticipated `CookieContainer` object sharing often results in incorrect code.

For more information about managing `HttpClient` lifetime with `IHttpClientFactory`, see [IHttpClientFactory guidelines](#).

Resilience policies with static clients

It's possible to configure a `static` or `singleton` client to use any number of resilience policies using the following pattern:

C#

```
using System;
using System.Net.Http;
using Microsoft.Extensions.Http;
using Polly;
using Polly.Extensions.Http;

var retryPolicy = HttpPolicyExtensions
    .HandleTransientHttpError()
    .WaitAndRetryAsync(3, retryAttempt => TimeSpan.FromSeconds(Math.Pow(2,
retryAttempt)));

var socketHandler = new SocketsHttpHandler { PooledConnectionLifetime =
TimeSpan.FromMinutes(15) };
var policyHandler = new PolicyHttpMessageHandler(retryPolicy)
{
    InnerHandler = socketHandler,
};

var httpClient = new HttpClient(policyHandler);
```

The preceding code:

- Relies on [Microsoft.Extensions.Http.Polly](#) NuGet package, transitively the [Polly.Extensions.Http](#) NuGet package for the `HttpPolicyExtensions` type.
- Specifies a transient HTTP error handler, configured with retry policy that with each attempt will exponentially backoff delay intervals.
- Defines a pooled connection lifetime of fifteen minutes for the `socketHandler`.
- Passes the `socketHandler` to the `policyHandler` with the retry logic.

- Instantiates an `HttpClient` given the `policyHandler`.

See also

- [HTTP support in .NET](#)
- [HTTP client factory with .NET](#)
- [Make HTTP requests with the HttpClient](#)
- [Use IHttpClientFactory to implement resilient HTTP requests](#)

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

Make HTTP requests with the HttpClient class

Article • 11/02/2023

In this article, you'll learn how to make HTTP requests and handle responses with the `HttpClient` class.

ⓘ Important

All of the example HTTP requests target one of the following URLs:

- <https://jsonplaceholder.typicode.com>: Free fake API for testing and prototyping.
- <https://www.example.com>: This domain is for use in illustrative examples in documents.

HTTP endpoints commonly return JavaScript Object Notation (JSON) data, but not always. For convenience, the optional [System.Net.Http.Json](#) NuGet package provides several extension methods for `HttpClient` and `HttpContent` that perform automatic serialization and deserialization using `System.Text.Json`. The examples that follow call attention to places where these extensions are available.

💡 Tip

All of the source code from this article is available in the [GitHub: .NET Docs](#) repository.

Create an `HttpClient`

Most of the following examples reuse the same `HttpClient` instance, and therefore only need to be configured once. To create an `HttpClient`, use the `HttpClient` class constructor. For more information, see [Guidelines for using HttpClient](#).

C#

```
// HttpClient lifecycle management best practices:  
//  
https://learn.microsoft.com/dotnet/fundamentals/networking/http/httpclient-guidelines#recommended-use
```

```
private static HttpClient sharedClient = new()
{
    BaseAddress = new Uri("https://jsonplaceholder.typicode.com"),
};
```

The preceding code:

- Instantiates a new `HttpClient` instance as a `static` variable. As per the [guidelines](#), it's recommended to reuse `HttpClient` instances during the application's lifecycle.
- Sets the `HttpClient.BaseAddress` to `"https://jsonplaceholder.typicode.com"`.

This `HttpClient` instance uses the base address when making subsequent requests. To apply other configuration, consider:

- Setting `HttpClient.DefaultRequestHeaders`.
- Applying a nondefault `HttpClient.Timeout`.
- Specifying the `HttpClient.DefaultRequestVersion`.

Tip

Alternatively, you can create `HttpClient` instances using a factory-pattern approach that allows you to configure any number of clients and consume them as dependency injection services. For more information, see [HTTP client factory with .NET](#).

Make an HTTP request

To make an HTTP request, you call any of the following APIs:

| HTTP method | API |
|-------------|--|
| GET | HttpClient.GetAsync |
| GET | HttpClient.GetByteArrayAsync |
| GET | HttpClient.GetStreamAsync |
| GET | HttpClient.GetStringAsync |
| POST | HttpClient.PostAsync |
| PUT | HttpClient.PutAsync |
| PATCH | HttpClient.PatchAsync |

| HTTP method | API |
|-----------------------------|-------------------------------------|
| DELETE | <code>HttpClient.DeleteAsync</code> |
| [†] USER SPECIFIED | <code>HttpClient.SendAsync</code> |

[†]A `USER SPECIFIED` request indicates that the `SendAsync` method accepts any valid `HttpMethod`.

⚠ Warning

Making HTTP requests is considered network I/O-bound work. While there is a synchronous `HttpClient.Send` method, it is recommended to use the asynchronous APIs instead, unless you have good reason not to.

HTTP content

The `HttpContent` type is used to represent an HTTP entity body and corresponding content headers. For HTTP methods (or request methods) that require a body, `POST`, `PUT`, and `PATCH`, you use the `HttpContent` class to specify the body of the request. Most examples show how to prepare the `StringContent` subclass with a JSON payload, but other subclasses exist for different [content \(MIME\) types](#).

- `ByteArrayContent`: Provides HTTP content based on a byte array.
- `FormUrlEncodedContent`: Provides HTTP content for name/value tuples encoded using `"application/x-www-form-urlencoded"` MIME type.
- `JsonContent`: Provides HTTP content based on JSON.
- `MultipartContent`: Provides a collection of `HttpContent` objects that get serialized using the `"multipart/*"` MIME type specification.
- `MultipartFormDataContent`: Provides a container for content encoded using `"multipart/form-data"` MIME type.
- `ReadOnlyMemoryContent`: Provides HTTP content based on a `ReadOnlyMemory<T>`.
- `StreamContent`: Provides HTTP content based on a stream.
- `StringContent`: Provides HTTP content based on a string.

The `HttpContent` class is also used to represent the response body of the `HttpResponseMessage`, accessible on the `HttpResponseMessage.Content` property.

HTTP Get

A `GET` request shouldn't send a body and is used (as the method name indicates) to retrieve (or get) data from a resource. To make an HTTP `GET` request, given an `HttpClient` and a URI, use the `HttpClient.GetAsync` method:

C#

```
static async Task GetAsync(HttpClient httpClient)
{
    using HttpResponseMessage response = await
httpClient.GetAsync("todos/3");

    response.EnsureSuccessStatusCode()
    .WriteRequestToConsole();

    var jsonResponse = await response.Content.ReadAsStringAsync();
    Console.WriteLine($"{jsonResponse}\n");

    // Expected output:
    //   GET https://jsonplaceholder.typicode.com/todos/3 HTTP/1.1
    //   {
    //     "userId": 1,
    //     "id": 3,
    //     "title": "fugiat veniam minus",
    //     "completed": false
    //   }
}
```

The preceding code:

- Makes a `GET` request to "`https://jsonplaceholder.typicode.com/todos/3`".
- Ensures that the response is successful.
- Writes the request details to the console.
- Reads the response body as a string.
- Writes the JSON response body to the console.

The `WriteRequestToConsole` is a custom extension method that isn't part of the framework, but if you're curious about how it's implemented, consider the following C# code:

C#

```
static class HttpResponseMessageExtensions
{
    internal static void WriteRequestToConsole(this HttpResponseMessage
response)
    {
        if (response is null)
        {
            return;
        }

        // Implementation of WriteRequestToConsole goes here
    }
}
```

```
        }

        var request = response.RequestMessage;
        Console.Write($"{request?.Method} ");
        Console.Write($"{request?.RequestUri} ");
        Console.WriteLine($"HTTP/{request?.Version}");

    }
}
```

This functionality is used to write the request details to the console in the following form:

```
<HTTP Request Method> <Request URI> <HTTP/Version>
```

As an example, the `GET` request to `https://jsonplaceholder.typicode.com/todos/3` outputs the following message:

Output

```
GET https://jsonplaceholder.typicode.com/todos/3 HTTP/1.1
```

HTTP Get from JSON

The <https://jsonplaceholder.typicode.com/todos> endpoint returns a JSON array of "todo" objects. Their JSON structure resembles the following:

JSON

```
[
  {
    "userId": 1,
    "id": 1,
    "title": "example title",
    "completed": false
  },
  {
    "userId": 1,
    "id": 2,
    "title": "another example title",
    "completed": true
  }
]
```

The C# `Todo` object is defined as follows:

C#

```
public record class Todo(
    int? UserId = null,
    int? Id = null,
    string? Title = null,
    bool? Completed = null);
```

It's a `record class` type, with optional `Id`, `Title`, `Completed`, and `UserId` properties. For more information on the `record` type, see [Introduction to record types in C#](#). To automatically deserialize `GET` requests into strongly-typed C# object, use the `GetFromJsonAsync` extension method that's part of the [System.Net.Http.Json](#) NuGet package.

C#

```
static async Task GetFromJsonAsync(HttpClient httpClient)
{
    var todos = await httpClient.GetFromJsonAsync<List<Todo>>(
        "todos?userId=1&completed=false");

    Console.WriteLine("GET https://jsonplaceholder.typicode.com/todos?
userId=1&completed=false HTTP/1.1");
    todos.ForEach(Console.WriteLine);
    Console.WriteLine();

    // Expected output:
    //   GET https://jsonplaceholder.typicode.com/todos?
    userId=1&completed=false HTTP/1.1
    //   Todo { UserId = 1, Id = 1, Title = delectus aut autem, Completed =
    False }
    //   Todo { UserId = 1, Id = 2, Title = quis ut nam facilis et officia
    qui, Completed = False }
    //   Todo { UserId = 1, Id = 3, Title = fugiat veniam minus, Completed =
    False }
    //   Todo { UserId = 1, Id = 5, Title = laboriosam mollitia et enim
    quasi adipisci quia provident illum, Completed = False }
    //   Todo { UserId = 1, Id = 6, Title = qui ullam ratione quibusdam
    voluptatem quia omnis, Completed = False }
    //   Todo { UserId = 1, Id = 7, Title = illo expedita consequatur quia
    in, Completed = False }
    //   Todo { UserId = 1, Id = 9, Title = molestiae perspiciatis ipsa,
    Completed = False }
    //   Todo { UserId = 1, Id = 13, Title = et doloremque nulla, Completed
    = False }
    //   Todo { UserId = 1, Id = 18, Title = dolorum est consequatur ea
    mollitia in culpa, Completed = False }
}
```

In the preceding code:

- A `GET` request is made to `"https://jsonplaceholder.typicode.com/todos?userId=1&completed=false"`.
 - The query string represents the filtering criteria for the request.
- The response is automatically deserialized into a `List<Todo>` when successful.
- The request details are written to the console, along with each `Todo` object.

HTTP Post

A `POST` request sends data to the server for processing. The `Content-Type` header of the request signifies what **MIME type** the body is sending. To make an HTTP `POST` request, given an `HttpClient` and a `Uri`, use the `HttpClient.PostAsync` method:

C#

```
static async Task PostAsync(HttpClient httpClient)
{
    using StringContent jsonContent = new(
        JsonSerializer.Serialize(new
        {
            userId = 77,
            id = 1,
            title = "write code sample",
            completed = false
        }),
        Encoding.UTF8,
        "application/json");

    using HttpResponseMessage response = await httpClient.PostAsync(
        "todos",
        jsonContent);

    response.EnsureSuccessStatusCode()
        .WriteRequestToConsole();

    var jsonResponse = await response.Content.ReadAsStringAsync();
    Console.WriteLine($"{jsonResponse}\n");

    // Expected output:
    // POST https://jsonplaceholder.typicode.com/todos HTTP/1.1
    // {
    //     "userId": 77,
    //     "id": 201,
    //     "title": "write code sample",
    //     "completed": false
    // }
}
```

The preceding code:

- Prepares a `StringContent` instance with the JSON body of the request (MIME type of `"application/json"`).
- Makes a `POST` request to `"https://jsonplaceholder.typicode.com/todos"`.
- Ensures that the response is successful, and writes the request details to the console.
- Writes the response body as a string to the console.

HTTP Post as JSON

To automatically serialize `POST` request arguments and deserialize responses into strongly-typed C# objects, use the `PostAsJsonAsync` extension method that's part of the [System.Net.Http.Json](#) NuGet package.

C#

```
static async Task PostAsJsonAsync(HttpClient httpClient)
{
    using HttpResponseMessage response = await httpClient.PostAsJsonAsync(
        "todos",
        new Todo(UserId: 9, Id: 99, Title: "Show extensions", Completed:
false));

    response.EnsureSuccessStatusCode()
        .WriteRequestToConsole();

    var todo = await response.Content.ReadFromJsonAsync<Todo>();
    Console.WriteLine($"{todo}\n");

    // Expected output:
    // POST https://jsonplaceholder.typicode.com/todos HTTP/1.1
    // Todo { UserId = 9, Id = 201, Title = Show extensions, Completed =
False }
}
```

The preceding code:

- Serializes the `Todo` instance as JSON, and makes a `POST` request to `"https://jsonplaceholder.typicode.com/todos"`.
- Ensures that the response is successful, and writes the request details to the console.
- Deserializes the response body into a `Todo` instance, and writes the `Todo` to the console.

HTTP Put

The `PUT` request method either replaces an existing resource or creates a new one using request body payload. To make an HTTP `PUT` request, given an `HttpClient` and a URI, use the `HttpClient.PutAsync` method:

C#

```
static async Task PutAsync(HttpClient httpClient)
{
    using StringContent jsonContent = new(
        JsonSerializer.Serialize(new
        {
            userId = 1,
            id = 1,
            title = "foo bar",
            completed = false
        }),
        Encoding.UTF8,
        "application/json");

    using HttpResponseMessage response = await httpClient.PutAsync(
        "todos/1",
        jsonContent);

    response.EnsureSuccessStatusCode()
        .WriteRequestToConsole();

    var jsonResponse = await response.Content.ReadAsStringAsync();
    Console.WriteLine($"{jsonResponse}\n");

    // Expected output:
    //   PUT https://jsonplaceholder.typicode.com/todos/1 HTTP/1.1
    //   {
    //     "userId": 1,
    //     "id": 1,
    //     "title": "foo bar",
    //     "completed": false
    //   }
}
```

The preceding code:

- Prepares a `StringContent` instance with the JSON body of the request (MIME type of `"application/json"`).
- Makes a `PUT` request to `"https://jsonplaceholder.typicode.com/todos/1"`.
- Ensures that the response is successful, and writes the request details and JSON response body to the console.

HTTP Put as JSON

To automatically serialize `PUT` request arguments and deserialize responses into strongly typed C# objects, use the [PutAsJsonAsync](#) extension method that's part of the [System.Net.Http.Json](#) NuGet package.

C#

```
static async Task PutAsJsonAsync(HttpClient httpClient)
{
    using HttpResponseMessage response = await httpClient.PutAsJsonAsync(
        "todos/5",
        new Todo(Title: "partially update todo", Completed: true));

    response.EnsureSuccessStatusCode()
        .WriteRequestToConsole();

    var todo = await response.Content.ReadFromJsonAsync<Todo>();
    Console.WriteLine($"{todo}\n");

    // Expected output:
    //   PUT https://jsonplaceholder.typicode.com/todos/5 HTTP/1.1
    //   Todo { UserId = , Id = 5, Title = partially update todo, Completed
    = True }
}
```

The preceding code:

- Serializes the `Todo` instance as JSON, and makes a `PUT` request to `"https://jsonplaceholder.typicode.com/todos/5"`.
- Ensures that the response is successful, and writes the request details to the console.
- Deserializes the response body into a `Todo` instance, and writes the `Todo` to the console.

HTTP Patch

The `PATCH` request is a partial update to an existing resource. It doesn't create a new resource, and it's not intended to replace an existing resource. Instead, it updates a resource only partially. To make an HTTP `PATCH` request, given an `HttpClient` and a URI, use the [HttpClient.PatchAsync](#) method:

C#

```
static async Task PatchAsync(HttpClient httpClient)
{
    using StringContent jsonContent = new(
        JsonSerializer.Serialize(new
    {
```

```

        completed = true
    }),
    Encoding.UTF8,
    "application/json");

    using HttpResponseMessage response = await httpClient.PatchAsync(
        "todos/1",
        jsonContent);

    response.EnsureSuccessStatusCode()
    .WriteRequestToConsole();

    var jsonResponse = await response.Content.ReadAsStringAsync();
    Console.WriteLine($"{{jsonResponse}}\n");

    // Expected output
    // PATCH https://jsonplaceholder.typicode.com/todos/1 HTTP/1.1
    // {
    //   "userId": 1,
    //   "id": 1,
    //   "title": "delectus aut autem",
    //   "completed": true
    // }
}

```

The preceding code:

- Prepares a `StringContent` instance with the JSON body of the request (MIME type of `"application/json"`).
- Makes a `PATCH` request to `"https://jsonplaceholder.typicode.com/todos/1"`.
- Ensures that the response is successful, and writes the request details and JSON response body to the console.

No extension methods exist for `PATCH` requests in the `System.Net.Http.Json` NuGet package.

HTTP Delete

A `DELETE` request deletes an existing resource. A `DELETE` request is *idempotent* but not *safe*, meaning multiple `DELETE` requests to the same resources yield the same result, but the request affects the state of the resource. To make an HTTP `DELETE` request, given an `HttpClient` and a URI, use the `HttpClient.DeleteAsync` method:

C#

```

static async Task DeleteAsync(HttpClient httpClient)
{
    using HttpResponseMessage response = await

```

```
httpClient.DeleteAsync("todos/1");

response.EnsureSuccessStatusCode()
    .WriteRequestToConsole();

var jsonResponse = await response.Content.ReadAsStringAsync();
Console.WriteLine($"{jsonResponse}\n");

// Expected output
// DELETE https://jsonplaceholder.typicode.com/todos/1 HTTP/1.1
// {}
}
```

The preceding code:

- Makes a `DELETE` request to `"https://jsonplaceholder.typicode.com/todos/1"`.
- Ensures that the response is successful, and writes the request details to the console.

Tip

The response to a `DELETE` request (just like a `PUT` request) may or may not include a body.

HTTP Head

The `HEAD` request is similar to a `GET` request. Instead of returning the resource, it only returns the headers associated with the resource. A response to the `HEAD` request doesn't return a body. To make an HTTP `HEAD` request, given an `HttpClient` and a URI, use the `HttpClient.SendAsync` method with the `HttpMethod` set to `HttpMethod.Head`:

C#

```
static async Task HeadAsync(HttpClient httpClient)
{
    using HttpRequestMessage request = new(
        HttpMethod.Head,
        "https://www.example.com");

    using HttpResponseMessage response = await
    httpClient.SendAsync(request);

    response.EnsureSuccessStatusCode()
        .WriteRequestToConsole();

    foreach (var header in response.Headers)
    {
```

```

        Console.WriteLine($"{header.Key}: {string.Join(", ", header.Value)}");
    }
    Console.WriteLine();

    // Expected output:
    // HEAD https://www.example.com/ HTTP/1.1
    // Accept-Ranges: bytes
    // Age: 550374
    // Cache-Control: max-age=604800
    // Date: Wed, 10 Aug 2022 17:24:55 GMT
    // ETag: "3147526947"
    // Server: ECS, (cha / 80E2)
    // X-Cache: HIT
}

```

The preceding code:

- Makes a `HEAD` request to `"https://www.example.com/"`.
- Ensures that the response is successful, and writes the request details to the console.
- Iterates over all of the response headers, writing each one to the console.

HTTP Options

The `OPTIONS` request is used to identify which HTTP methods a server or endpoint supports. To make an HTTP `OPTIONS` request, given an `HttpClient` and a URI, use the `HttpClient.SendAsync` method with the `HttpMethod` set to `HttpMethod.Options`:

C#

```

static async Task OptionsAsync(HttpClient httpClient)
{
    using HttpRequestMessage request = new(
        HttpMethod.Options,
        "https://www.example.com");

    using HttpResponseMessage response = await
    httpClient.SendAsync(request);

    response.EnsureSuccessStatusCode()
        .WriteRequestToConsole();

    foreach (var header in response.Content.Headers)
    {
        Console.WriteLine($"{header.Key}: {string.Join(", ", header.Value)}");
    }
    Console.WriteLine();
}

```

```
// Expected output
//   OPTIONS https://www.example.com/ HTTP/1.1
//   Allow: OPTIONS, GET, HEAD, POST
//   Content-Type: text/html; charset=utf-8
//   Expires: Wed, 17 Aug 2022 17:28:42 GMT
//   Content-Length: 0
}
```

The preceding code:

- Sends an `OPTIONS` HTTP request to `"https://www.example.com/"`.
- Ensures that the response is successful, and writes the request details to the console.
- Iterates over all of the response content headers, writing each one to the console.

HTTP Trace

The `TRACE` request can be useful for debugging as it provides application-level loop-back of the request message. To make an HTTP `TRACE` request, create an `HttpRequestMessage` using the `HttpMethod.Trace`:

C#

```
using HttpRequestMessage request = new(
    HttpMethod.Trace,
    "{ValidRequestUri}");
```

⊗ Caution

The `TRACE` HTTP method is not supported by all HTTP servers. It can expose a security vulnerability if used unwisely. For more information, see [Open Web Application Security Project \(OWASP\): Cross Site Tracing](#).

Handle an HTTP response

Whenever you're handling an HTTP response, you interact with the `HttpResponseMessage` type. Several members are used when evaluating the validity of a response. The HTTP status code is available via the `HttpResponseMessage.StatusCode` property. Imagine that you've sent a request given a client instance:

C#

```
using HttpResponseMessage response = await httpClient.SendAsync(request);
```

To ensure that the `response` is `OK` (HTTP status code 200), you can evaluate it as shown in the following example:

C#

```
if (response is { StatusCode: HttpStatusCode.OK })
{
    // Omitted for brevity...
}
```

There are other HTTP status codes that represent a successful response, such as `CREATED` (HTTP status code 201), `ACCEPTED` (HTTP status code 202), `NO CONTENT` (HTTP status code 204), and `RESET CONTENT` (HTTP status code 205). You can use the [HttpResponseMessage.IsSuccessStatusCode](#) property to evaluate these codes as well, which ensures that the response status code is within the range 200-299:

C#

```
if (response.IsSuccessStatusCode)
{
    // Omitted for brevity...
}
```

If you need to have the framework throw the [HttpRequestException](#), you can call the [HttpResponseMessage.EnsureSuccessStatusCode\(\)](#) method:

C#

```
response.EnsureSuccessStatusCode();
```

This code throws an `HttpRequestException` if the response status code isn't within the 200-299 range.

HTTP valid content responses

With a valid response, you can access the response body using the [Content](#) property. The body is available as an [HttpContent](#) instance, which you can use to access the body as a stream, byte array, or string:

C#

```
await using Stream responseStream =
    await response.Content.ReadAsStreamAsync();
```

In the preceding code, the `responseStream` can be used to read the response body.

C#

```
byte[] responseByteArray = await response.Content.ReadAsByteArrayAsync();
```

In the preceding code, the `responseByteArray` can be used to read the response body.

C#

```
string responseString = await response.Content.ReadAsStringAsync();
```

In the preceding code, the `responseString` can be used to read the response body.

Finally, when you know an HTTP endpoint returns JSON, you can deserialize the response body into any valid C# object by using the [System.Net.Http.Json](#) NuGet package:

C#

```
T? result = await response.Content.ReadFromJsonAsync<T>();
```

In the preceding code, `result` is the response body deserialized as the type `T`.

HTTP error handling

When an HTTP request fails, the [HttpRequestException](#) is thrown. Catching that exception alone may not be sufficient, as there are other potential exceptions thrown that you might want to consider handling. For example, the calling code may have used a cancellation token that was canceled before the request was completed. In this scenario, you'd catch the [TaskCanceledException](#):

C#

```
using var cts = new CancellationTokenSource();
try
{
    // Assuming:
    // httpClient.Timeout = TimeSpan.FromSeconds(10)
```

```
        using var response = await httpClient.GetAsync(
            "http://localhost:5001/sleepFor?seconds=100", cts.Token);
    }
    catch (OperationCanceledException ex) when (cts.IsCancellationRequested)
    {
        // When the token has been canceled, it is not a timeout.
        Console.WriteLine($"Canceled: {ex.Message}");
    }
}
```

Likewise, when making an HTTP request, if the server doesn't respond before the `HttpClient.Timeout` is exceeded the same exception is thrown. However, in this scenario, you can distinguish that the timeout occurred by evaluating the `Exception.InnerException` when catching the `TaskCanceledException`:

C#

```
try
{
    // Assuming:
    //   httpClient.Timeout = TimeSpan.FromSeconds(10)

    using var response = await httpClient.GetAsync(
        "http://localhost:5001/sleepFor?seconds=100");
}
catch (OperationCanceledException ex) when (ex.InnerException is
TimeoutException tex)
{
    Console.WriteLine($"Timed out: {ex.Message}, {tex.Message}");
}
```

In the preceding code, when the inner exception is a `TimeoutException` the timeout occurred, and the request wasn't canceled by the cancellation token.

To evaluate the HTTP status code when catching an `HttpRequestException`, you can evaluate the `HttpRequestException.StatusCode` property:

C#

```
try
{
    // Assuming:
    //   httpClient.Timeout = TimeSpan.FromSeconds(10)

    using var response = await httpClient.GetAsync(
        "http://localhost:5001/doesNotExist");

    response.EnsureSuccessStatusCode();
}
catch (HttpRequestException ex) when (ex is { StatusCode:
    HttpStatusCode.NotFound })
```

```
{  
    // Handle 404  
    Console.WriteLine($"Not found: {ex.Message}");  
}
```

In the preceding code, the `EnsureSuccessStatusCode()` method is called to throw an exception if the response isn't successful. The `HttpRequestException.StatusCode` property is then evaluated to determine if the response was a `404` (HTTP status code 404). There are several helper methods on `HttpClient` that implicitly call `EnsureSuccessStatusCode` on your behalf, consider the following APIs:

- `HttpClient.GetByteArrayAsync`
- `HttpClient.GetStreamAsync`
- `HttpClient.GetStringAsync`

💡 Tip

All `HttpClient` methods used to make HTTP requests that don't return an `HttpResponseMessage` implicitly call `EnsureSuccessStatusCode` on your behalf.

When calling these methods, you can handle the `HttpRequestException` and evaluate the `HttpRequestException.StatusCode` property to determine the HTTP status code of the response:

C#

```
try  
{  
    // These extension methods will throw HttpRequestException  
    // with StatusCode set when the HTTP request status code isn't 2xx:  
    //  
    //    GetByteArrayAsync  
    //    GetStreamAsync  
    //    GetStringAsync  
  
    using var stream = await httpClient.GetStreamAsync(  
        "https://localhost:5001/doesNotExist");  
}  
catch (HttpRequestException ex) when (ex is { StatusCode:  
    HttpStatusCode.NotFound })  
{  
    // Handle 404  
    Console.WriteLine($"Not found: {ex.Message}");  
}
```

There might be scenarios in which you need to throw the [HttpRequestException](#) in your code. The [HttpRequestException\(\)](#) constructor is public, and you can use it to throw an exception with a custom message:

```
C#  
  
try  
{  
    using var response = await httpClient.GetAsync(  
        "https://localhost:5001/doesNotExists");  
  
    // Throw for anything higher than 400.  
    if (response is { StatusCode: >= HttpStatusCode.BadRequest })  
    {  
        throw new HttpRequestException(  
            "Something went wrong", inner: null, response.StatusCode);  
    }  
}  
catch (HttpRequestException ex) when (ex is { StatusCode:  
HttpStatusCode.NotFound })  
{  
    Console.WriteLine($"Not found: {ex.Message}");  
}
```

HTTP proxy

An HTTP proxy can be configured in one of two ways. A default is specified on the [HttpClient.DefaultProxy](#) property. Alternatively, you can specify a proxy on the [HttpClientHandler.Proxy](#) property.

Global default proxy

The [HttpClient.DefaultProxy](#) is a static property that determines the default proxy that all [HttpClient](#) instances use if no proxy is set explicitly in the [HttpClientHandler](#) passed through its constructor.

The default instance returned by this property initializes following a different set of rules depending on your platform:

- **For Windows:** Reads proxy configuration from environment variables or, if those aren't defined, from the user's proxy settings.
- **For macOS:** Reads proxy configuration from environment variables or, if those aren't defined, from the system's proxy settings.
- **For Linux:** Reads proxy configuration from environment variables or, in case those aren't defined, this property initializes a non-configured instance that bypasses all

addresses.

The environment variables used for `DefaultProxy` initialization on Windows and Unix-based platforms are:

- `HTTP_PROXY`: the proxy server used on HTTP requests.
- `HTTPS_PROXY`: the proxy server used on HTTPS requests.
- `ALL_PROXY`: the proxy server used on HTTP and/or HTTPS requests in case `HTTP_PROXY` and/or `HTTPS_PROXY` aren't defined.
- `NO_PROXY`: a comma-separated list of hostnames that should be excluded from proxying. Asterisks aren't supported for wildcards; use a leading dot in case you want to match a subdomain. Examples: `NO_PROXY=.example.com` (with leading dot) will match `www.example.com`, but won't match `example.com`. `NO_PROXY=example.com` (without leading dot) won't match `www.example.com`. This behavior might be revisited in the future to match other ecosystems better.

On systems where environment variables are case-sensitive, the variable names may be all lowercase or all uppercase. The lowercase names are checked first.

The proxy server may be a hostname or IP address, optionally followed by a colon and port number, or it may be an `http` URL, optionally including a username and password for proxy authentication. The URL must be start with `http`, not `https`, and can't include any text after the hostname, IP, or port.

Proxy per client

The `HttpClientHandler.Proxy` property identifies the `WebProxy` object to use to process requests to Internet resources. To specify that no proxy should be used, set the `Proxy` property to the proxy instance returned by the `GlobalProxySelection.GetEmptyWebProxy()` method.

The local computer or application config file may specify that a default proxy is used. If the `Proxy` property is specified, then the proxy settings from the `Proxy` property override the local computer or application config file and the handler uses the proxy settings specified. If no proxy is specified in a config file and the `Proxy` property is unspecified, the handler uses the proxy settings inherited from the local computer. If there are no proxy settings, the request is sent directly to the server.

The `HttpClientHandler` class parses a proxy bypass list with wildcard characters inherited from local computer settings. For example, the `HttpClientHandler` class parses a bypass

list of "nt*" from browsers as a regular expression of "nt.*". So a URL of `http://nt.com` would bypass the proxy using the `HttpClientHandler` class.

The `HttpClientHandler` class supports local proxy bypass. The class considers a destination to be local if any of the following conditions are met:

1. The destination contains a flat name (no dots in the URL).
2. The destination contains a loopback address ([Loopback](#) or [IPv6Loopback](#)) or the destination contains an [IPAddress](#) assigned to the local computer.
3. The domain suffix of the destination matches the local computer's domain suffix ([DomainName](#)).

For more information about configuring a proxy, see:

- [WebProxy.Address](#)
- [WebProxy.BypassProxyOnLocal](#)
- [WebProxy.BypassArrayList](#)

See also

- [HTTP support in .NET](#)
- [Guidelines for using HttpClient](#)
- [HTTP client factory with .NET](#)
- [Use HTTP/3 with HttpClient](#)
- [Test web APIs with the HttpRepl](#)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

IHttpClientFactory with .NET

Article • 05/19/2023

In this article, you'll learn how to use the `IHttpClientFactory` to create `HttpClient` types with various .NET fundamentals, such as dependency injection (DI), logging, and configuration. The `HttpClient` type was introduced in .NET Framework 4.5, which was released in 2012. In other words, it's been around for a while. `HttpClient` is used for making HTTP requests and handling HTTP responses from web resources identified by a `Uri`. The HTTP protocol makes up the vast majority of all internet traffic.

With modern application development principles driving best practices, the `IHttpClientFactory` serves as a factory abstraction that can create `HttpClient` instances with custom configurations. `IHttpClientFactory` was introduced in .NET Core 2.1. Common HTTP-based .NET workloads can take advantage of resilient and transient-fault-handling third-party middleware with ease.

ⓘ Note

If your app requires cookies, it might be better to avoid using `IHttpClientFactory` in your app. For alternative ways of managing clients, see [Guidelines for using HTTP clients](#).

ⓘ Important

Lifetime management of `HttpClient` instances created by `IHttpClientFactory` is completely different from instances created manually. The strategies are to use either **short-lived** clients created by `IHttpClientFactory` or **long-lived** clients with `PooledConnectionLifetime` set up. For more information, see the [HttpClient lifetime management](#) section and [Guidelines for using HTTP clients](#).

The `IHttpClientFactory` type

All of the sample source code in this article relies on the [Microsoft.Extensions.Http](#) NuGet package. Additionally, HTTP `GET` requests are made to the free [{JSON} Placeholder](#) API to get user `Todo` objects.

When you call any of the `AddHttpClient` extension methods, you're adding the `IHttpClientFactory` and related services to the `IServiceCollection`. The

`IHttpClientFactory` type offers the following benefits:

- Exposes the `HttpClient` class as a DI-ready type.
- Provides a central location for naming and configuring logical `HttpClient` instances.
- Codifies the concept of outgoing middleware via delegating handlers in `HttpClient`.
- Provides extension methods for Polly-based middleware to take advantage of delegating handlers in `HttpClient`.
- Manages the caching and lifetime of underlying `HttpClientHandler` instances. Automatic management avoids common Domain Name System (DNS) problems that occur when manually managing `HttpClient` lifetimes.
- Adds a configurable logging experience (via `ILogger`) for all requests sent through clients created by the factory.

Consumption patterns

There are several ways `IHttpClientFactory` can be used in an app:

- [Basic usage](#)
- [Named clients](#)
- [Typed clients](#)
- [Generated clients](#)

The best approach depends upon the app's requirements.

Basic usage

To register the `IHttpClientFactory`, call `AddHttpClient`:

C#

```
using Shared;
using BasicHttp.Example;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
using Microsoft.Extensions.Logging;

HostApplicationBuilder builder = Host.CreateApplicationBuilder(args);

builder.Services.AddHttpClient();
builder.Services.AddTransient<TodoService>();
```

```
using IHost host = builder.Build();
```

Consuming services can require the `IHttpClientFactory` as a constructor parameter with DI. The following code uses `IHttpClientFactory` to create an `HttpClient` instance:

C#

```
using System.Net.Http.Json;
using System.Text.Json;
using Microsoft.Extensions.Logging;
using Shared;

namespace BasicHttp.Example;

public sealed class TodoService(
    IHttpClientFactory httpClientFactory,
    ILogger<TodoService> logger)
{
    public async Task<Todo[]> GetUserTodosAsync(int userId)
    {
        // Create the client
        using HttpClient client = httpClientFactory.CreateClient();

        try
        {
            // Make HTTP GET request
            // Parse JSON response deserialize into Todo types
            Todo[]? todos = await client.GetFromJsonAsync<Todo[]>(
                $"https://jsonplaceholder.typicode.com/todos?userId={userId}",
                new JsonSerializerOptions(JsonSerializerDefaults.Web));

            return todos ?? [];
        }
        catch (Exception ex)
        {
            logger.LogError("Error getting something fun to say: {Error}", ex);
        }

        return [];
    }
}
```

Using `IHttpClientFactory` like in the preceding example is a good way to refactor an existing app. It has no impact on how `HttpClient` is used. In places where `HttpClient` instances are created in an existing app, replace those occurrences with calls to [CreateClient](#).

Named clients

Named clients are a good choice when:

- The app requires many distinct uses of `HttpClient`.
- Many `HttpClient` instances have different configurations.

Configuration for a named `HttpClient` can be specified during registration on the `IServiceCollection`:

C#

```
using Shared;
using NamedHttp.Example;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
using Microsoft.Extensions.Logging;

HostApplicationBuilder builder = Host.CreateApplicationBuilder(args);

string? httpClientName = builder.Configuration[ "TodoHttpClientName" ];
ArgumentException.ThrowIfNullOrEmpty(httpClientName);

builder.Services.AddHttpClient(
    httpClientName,
    client =>
{
    // Set the base address of the named client.
    client.BaseAddress = new
Uri("https://jsonplaceholder.typicode.com/");

    // Add a user-agent default request header.
    client.DefaultRequestHeaders.UserAgent.ParseAdd( "dotnet-docs" );
});
```

In the preceding code, the client is configured with:

- A name that's pulled from the configuration under the `"TodoHttpClientName"`.
- The base address `https://jsonplaceholder.typicode.com/`.
- A `"User-Agent"` header.

You can use configuration to specify HTTP client names, which is helpful to avoid misnaming clients when adding and creating. In this example, the `appsettings.json` file is used to configure the HTTP client name:

JSON

```
{  
    "TodoHttpClientName": "JsonPlaceholderApi"  
}
```

It's easy to extend this configuration and store more details about how you'd like your HTTP client to function. For more information, see [Configuration in .NET](#).

Create client

Each time `CreateClient` is called:

- A new instance of `HttpClient` is created.
- The configuration action is called.

To create a named client, pass its name into `CreateClient`:

C#

```
using System.Net.Http.Json;  
using System.Text.Json;  
using Microsoft.Extensions.Configuration;  
using Microsoft.Extensions.Logging;  
using Shared;  
  
namespace NamedHttp.Example;  
  
public sealed class TodoService  
{  
    private readonly IHttpClientFactory _httpClientFactory = null!;  
    private readonly IConfiguration _configuration = null!;  
    private readonly ILogger<TodoService> _logger = null!;  
  
    public TodoService(  
        IHttpClientFactory httpClientFactory,  
        IConfiguration configuration,  
        ILogger<TodoService> logger) =>  
        (_httpClientFactory, _configuration, _logger) =  
            (httpClientFactory, configuration, logger);  
  
    public async Task<Todo[]> GetUserTodosAsync(int userId)  
    {  
        // Create the client  
        string? httpClientName = _configuration["TodoHttpClientName"];  
        using HttpClient client =  
            _httpClientFactory.CreateClient(httpClientName ?? "");  
  
        try  
        {  
            // Make HTTP GET request  
        }
```

```

        // Parse JSON response deserialize into Todo type
        Todo[]? todos = await client.GetFromJsonAsync<Todo[]>(
            $"todos?userId={userId}",
            new JsonSerializerOptions(JsonSerializerDefaults.Web));

        return todos ?? [];
    }
    catch (Exception ex)
    {
        _logger.LogError("Error getting something fun to say: {Error}",
ex);
    }

    return [];
}
}

```

In the preceding code, the HTTP request doesn't need to specify a hostname. The code can pass just the path, since the base address configured for the client is used.

Typed clients

Typed clients:

- Provide the same capabilities as named clients without the need to use strings as keys.
- Provide [IntelliSense](#) and compiler help when consuming clients.
- Provide a single location to configure and interact with a particular `HttpClient`. For example, a single typed client might be used:
 - For a single backend endpoint.
 - To encapsulate all logic dealing with the endpoint.
- Work with DI and can be injected where required in the app.

A typed client accepts an `HttpClient` parameter in its constructor:

C#

```

using System.Net.Http.Json;
using System.Text.Json;
using Microsoft.Extensions.Logging;
using Shared;

namespace TypedHttp.Example;

public sealed class TodoService(
    HttpClient httpClient,
    ILogger<TodoService> logger) : IDisposable
{

```

```

public async Task<Todo[]> GetUserTodosAsync(int userId)
{
    try
    {
        // Make HTTP GET request
        // Parse JSON response deserialize into Todo type
        Todo[]? todos = await httpClient.GetFromJsonAsync<Todo[]>(
            $"todos?userId={userId}",
            new JsonSerializerOptions(JsonSerializerDefaults.Web));

        return todos ?? [];
    }
    catch (Exception ex)
    {
        logger.LogError("Error getting something fun to say: {Error}",
ex);
    }

    return [];
}

public void Dispose() => httpClient?.Dispose();
}

```

In the preceding code:

- The configuration is set when the typed client is added to the service collection.
- The `HttpClient` is assigned as a class-scoped variable (field), and used with exposed APIs.

API-specific methods can be created that expose `HttpClient` functionality. For example, the `GetUserTodosAsync` method encapsulates code to retrieve user-specific `Todo` objects.

The following code calls `AddHttpClient` to register a typed client class:

C#

```

using Shared;
using TypedHttp.Example;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
using Microsoft.Extensions.Logging;

HostApplicationBuilder builder = Host.CreateApplicationBuilder(args);

```

```
builder.Services.AddHttpClient<TodoService>(
    client =>
{
    // Set the base address of the typed client.
    client.BaseAddress = new
Uri("https://jsonplaceholder.typicode.com/");

    // Add a user-agent default request header.
    client.DefaultRequestHeaders.UserAgent.ParseAdd("dotnet-docs");
});
```

The typed client is registered as transient with DI. In the preceding code, `AddHttpClient` registers `TodoService` as a transient service. This registration uses a factory method to:

1. Create an instance of `HttpClient`.
2. Create an instance of `TodoService`, passing in the instance of `HttpClient` to its constructor.

ⓘ Important

Using typed clients in singleton services can be dangerous. For more information, see the [Avoid Typed clients in singleton services](#) section.

ⓘ Note

When registering a typed client with the `AddHttpClient<TClient>` method, the `TClient` type must have a constructor that accepts an `HttpClient` parameter. Additionally, the `TClient` type shouldn't be registered with the DI container separately.

Generated clients

`IHttpClientFactory` can be used in combination with third-party libraries such as [Refit](#). Refit is a REST library for .NET. It allows for declarative REST API definitions, mapping interface methods to endpoints. An implementation of the interface is generated dynamically by the `RestService`, using `HttpClient` to make the external HTTP calls.

Consider the following `record` type:

C#

```
namespace Shared;

public record class Todo(
    int UserId,
    int Id,
    string Title,
    bool Completed);
```

The following example relies on the [Refit.HttpClientFactory](#) NuGet package, and is a simple interface:

C#

```
using Refit;
using Shared;

namespace GeneratedHttp.Example;

public interface ITodoService
{
    [Get("/todos?userId={userId}")]
    Task<Todo[]> GetUserTodosAsync(int userId);
}
```

The preceding C# interface:

- Defines a method named `GetUserTodosAsync` that returns a `Task<Todo[]>` instance.
- Declares a `Refit.GetAttribute` attribute with the path and query string to the external API.

A typed client can be added, using Refit to generate the implementation:

C#

```
using GeneratedHttp.Example;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
using Microsoft.Extensions.Logging;
using Refit;
using Shared;

HostApplicationBuilder builder = Host.CreateApplicationBuilder(args);
```

```
builder.Services.AddRefitClient<ITodoService>()
    .ConfigureHttpClient(client =>
{
    // Set the base address of the named client.
    client.BaseAddress = new
Uri("https://jsonplaceholder.typicode.com/");

    // Add a user-agent default request header.
    client.DefaultRequestHeaders.UserAgent.ParseAdd("dotnet-docs");
});
```

The defined interface can be consumed where necessary, with the implementation provided by DI and Refit.

Make POST, PUT, and DELETE requests

In the preceding examples, all HTTP requests use the `GET` HTTP verb. `HttpClient` also supports other HTTP verbs, including:

- `POST`
- `PUT`
- `DELETE`
- `PATCH`

For a complete list of supported HTTP verbs, see [HttpMethod](#). For more information on making HTTP requests, see [Send a request using HttpClient](#).

The following example shows how to make an HTTP `POST` request:

C#

```
public async Task CreateItemAsync(Item item)
{
    using StringContent json = new(
        JsonSerializer.Serialize(item, new
JsonSerializerOptions(JsonSerializerDefaults.Web)),
        Encoding.UTF8,
        MediaTypeNames.Application.Json);

    using HttpResponseMessage httpResponse =
        await httpClient.PostAsync("/api/items", json);

    httpResponse.EnsureSuccessStatusCode();
}
```

In the preceding code, the `CreateItemAsync` method:

- Serializes the `Item` parameter to JSON using `System.Text.Json`. This uses an instance of `JsonSerializerOptions` to configure the serialization process.
- Creates an instance of `StringContent` to package the serialized JSON for sending in the HTTP request's body.
- Calls `PostAsync` to send the JSON content to the specified URL. This is a relative URL that gets added to the `HttpClient.BaseAddress`.
- Calls `EnsureSuccessStatusCode` to throw an exception if the response status code does not indicate success.

`HttpClient` also supports other types of content. For example, `MultipartContent` and `StreamContent`. For a complete list of supported content, see [HttpContent](#).

The following example shows an HTTP `PUT` request:

C#

```
public async Task UpdateItemAsync(Item item)
{
    using StringContent json = new(
        JsonSerializer.Serialize(item, new
JsonSerializerOptions(JsonSerializerDefaults.Web)),
        Encoding.UTF8,
        MediaTypeNames.Application.Json);

    using HttpResponseMessage httpResponse =
        await httpClient.PutAsync($""/api/items/{item.Id}", json);

    httpResponse.EnsureSuccessStatusCode();
}
```

The preceding code is very similar to the `POST` example. The `UpdateItemAsync` method calls `PutAsync` instead of `PostAsync`.

The following example shows an HTTP `DELETE` request:

C#

```
public async Task DeleteItemAsync(Guid id)
{
    using HttpResponseMessage httpResponse =
        await httpClient.DeleteAsync($""/api/items/{id}");

    httpResponse.EnsureSuccessStatusCode();
}
```

In the preceding code, the `DeleteItemAsync` method calls `DeleteAsync`. Because HTTP DELETE requests typically contain no body, the `DeleteAsync` method doesn't provide an overload that accepts an instance of `HttpContent`.

To learn more about using different HTTP verbs with `HttpClient`, see [HttpClient](#).

HttpClient lifetime management

A new `HttpClient` instance is returned each time `createClient` is called on the `IHttpClientFactory`. One `HttpClientHandler` instance is created per client name. The factory manages the lifetimes of the `HttpClientHandler` instances.

`IHttpClientFactory` caches the `HttpClientHandler` instances created by the factory to reduce resource consumption. An `HttpClientHandler` instance may be reused from the cache when creating a new `HttpClient` instance if its lifetime hasn't expired.

Caching of handlers is desirable as each handler typically manages its own underlying HTTP connection pool. Creating more handlers than necessary can result in socket exhaustion and connection delays. Some handlers also keep connections open indefinitely, which can prevent the handler from reacting to DNS changes.

The default handler lifetime is two minutes. To override the default value, call `SetHandlerLifetime` for each client, on the `IServiceCollection`:

```
C#  
  
services.AddHttpClient("Named.Client")  
    .SetHandlerLifetime(TimeSpan.FromMinutes(5));
```

ⓘ Important

`HttpClient` instances created by `IHttpClientFactory` are intended to be **short-lived**.

- Recycling and recreating `HttpMessageHandler`'s when their lifetime expires is essential for `IHttpClientFactory` to ensure the handlers react to DNS changes. `HttpClient` is tied to a specific handler instance upon its creation, so new `HttpClient` instances should be requested in a timely manner to ensure the client will get the updated handler.

- Disposing of such `HttpClient` instances **created by the factory** will not lead to socket exhaustion, as its disposal **will not** trigger disposal of the `HttpMessageHandler`. `IHttpClientFactory` tracks and disposes of resources used to create `HttpClient` instances, specifically the `HttpMessageHandler` instances, as soon their lifetime expires and there's no `HttpClient` using them anymore.

Keeping a single `HttpClient` instance alive for a long duration is a common pattern that can be used as an **alternative** to `IHttpClientFactory`, however, this pattern requires additional setup, such as `PooledConnectionLifetime`. You can use either **long-lived** clients with `PooledConnectionLifetime`, or **short-lived** clients created by `IHttpClientFactory`. For information about which strategy to use in your app, see [Guidelines for using HTTP clients](#).

Configure the `HttpMessageHandler`

It may be necessary to control the configuration of the inner `HttpMessageHandler` used by a client.

An `IHttpClientBuilder` is returned when adding named or typed clients. The `ConfigurePrimaryHttpMessageHandler` extension method can be used to define a delegate on the `IServiceCollection`. The delegate is used to create and configure the primary `HttpMessageHandler` used by that client:

```
C#  
  
.ConfigurePrimaryHttpMessageHandler(() =>  
{  
    return new HttpClientHandler  
    {  
        AllowAutoRedirect = false,  
        UseDefaultCredentials = true  
    };  
});
```

Configuring the `HttpClientHandler` lets you specify a proxy for the `HttpClient` instance among various other properties of the handler. For more information, see [Proxy per client](#).

Additional configuration

There are several additional configuration options for controlling the

`IHttpClientHandler`:

[+] Expand table

| Method | Description |
|--|--|
| AddHttpMessageHandler | Adds an additional message handler for a named <code>HttpClient</code> . |
| AddTypedClient | Configures the binding between the <code>TClient</code> and the named <code>HttpClient</code> associated with the <code>IHttpClientBuilder</code> . |
| ConfigureHttpClient | Adds a delegate that will be used to configure a named <code>HttpClient</code> . |
| ConfigureHttpMessageHandlerBuilder | Adds a delegate that will be used to configure message handlers using <code>HttpMessageHandlerBuilder</code> for a named <code>HttpClient</code> . |
| ConfigurePrimaryHttpMessageHandler | Configures the primary <code>HttpMessageHandler</code> from the dependency injection container for a named <code>HttpClient</code> . |
| RedactLoggedHeaders | Sets the collection of HTTP header names for which values should be redacted before logging. |
| SetHandlerLifetime | Sets the length of time that a <code>HttpMessageHandler</code> instance can be reused. Each named client can have its own configured handler lifetime value. |

Using `IHttpClientFactory` together with `SocketsHttpHandler`

The `SocketsHttpHandler` implementation of `HttpMessageHandler` was added in .NET Core 2.1, which allows `PooledConnectionLifetime` to be configured. This setting is used to ensure that the handler reacts to DNS changes, so using `SocketsHttpHandler` is considered to be an alternative to using `IHttpClientFactory`. For more information, see [Guidelines for using HTTP clients](#).

However, `SocketsHttpHandler` and `IHttpClientFactory` can be used together to improve configurability. By using both of these APIs, you benefit from configurability on both a low level (for example, using `LocalCertificateSelectionCallback` for dynamic certificate selection) and a high level (for example, leveraging DI integration and several client configurations).

To use both APIs:

1. Specify `SocketsHttpHandler` as `PrimaryHandler` and set up its `PooledConnectionLifetime` (for example, to a value that was previously in `HandlerLifetime`).
2. As `SocketsHttpHandler` will handle connection pooling and recycling, then handler recycling at the `IHttpClientFactory` level is not needed anymore. You can disable it by setting `HandlerLifetime` to `Timeout.InfiniteTimeSpan`.

C#

```
services.AddHttpClient(name)
    .ConfigurePrimaryHttpMessageHandler(() =>
{
    return new SocketsHttpHandler()
    {
        PooledConnectionLifetime = TimeSpan.FromMinutes(2)
    };
})
    .SetHandlerLifetime(Timeout.InfiniteTimeSpan); // Disable rotation, as
it is handled by PooledConnectionLifetime
```

Avoid typed clients in singleton services

When using the *named client* approach, `IHttpClientFactory` is injected into services, and `HttpClient` instances are created by calling `CreateClient` every time an `HttpClient` is needed.

However, with the *typed client* approach, typed clients are transient objects usually injected into services. That may cause a problem because a typed client can be injected into a singleton service.

ⓘ Important

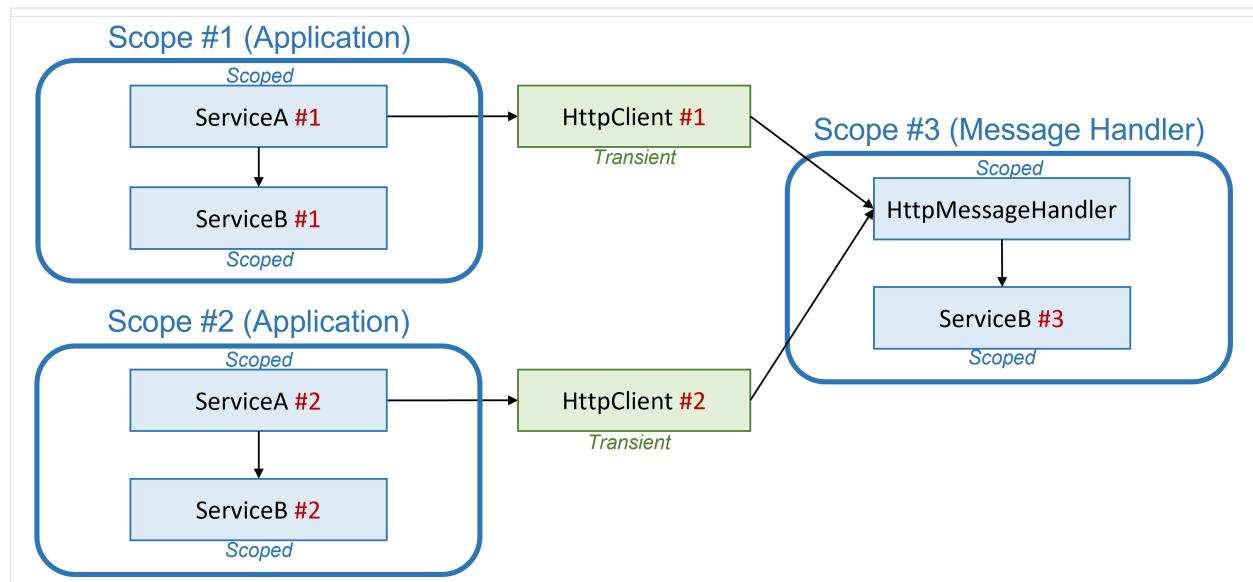
Typed clients are expected to be **short-lived** in the same sense as `HttpClient` instances created by `IHttpClientFactory` (for more information, see [HttpClient lifetime management](#)). As soon as a typed client instance is created, `IHttpClientFactory` has no control over it. If a typed client instance is captured in a singleton, it may prevent it from reacting to DNS changes, defeating one of the purposes of `IHttpClientFactory`.

If you need to use `HttpClient` instances in a singleton service, consider the following options:

- Use the *named client* approach instead, injecting `IHttpClientFactory` in the singleton service and recreating `HttpClient` instances when necessary.
- If you require the *typed client* approach, use `SocketsHttpHandler` with configured `PooledConnectionLifetime` as a primary handler. For more information on using `SocketsHttpHandler` with `IHttpClientFactory`, see the section [Using `IHttpClientFactory` together with `SocketsHttpHandler`](#).

Message Handler Scopes in `IHttpClientFactory`

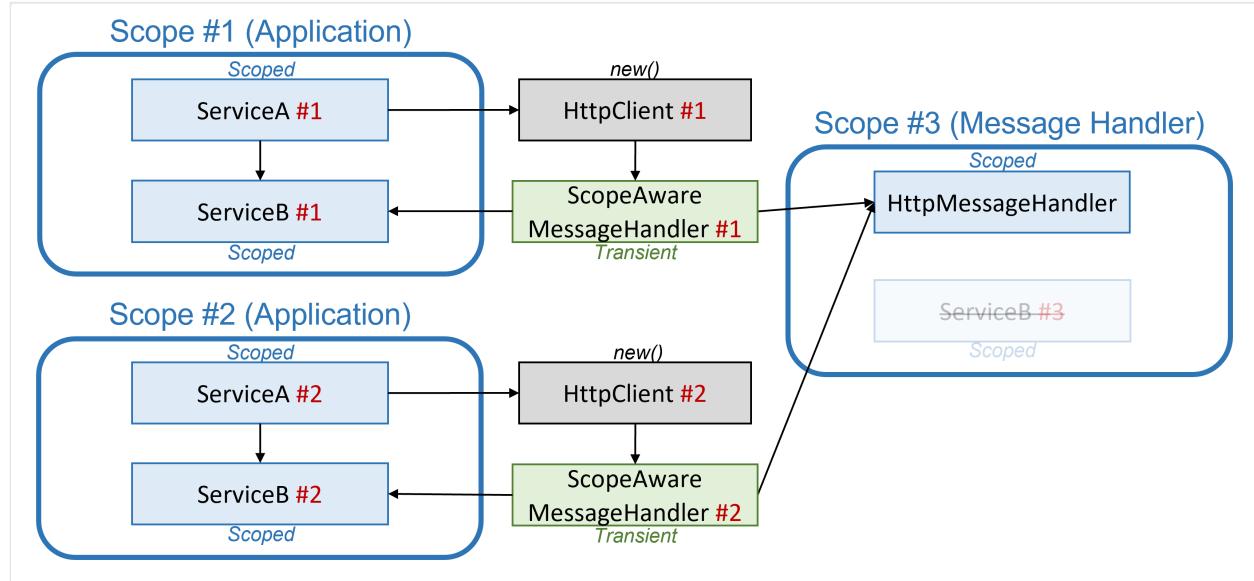
`IHttpClientFactory` creates a separate DI scope per each `HttpMessageHandler` instance. These DI scopes are separate from application DI scopes (for example, ASP.NET incoming request scope, or a user-created manual DI scope), so they will **not** share scoped service instances. Message Handler scopes are tied to handler lifetime and can outlive application scopes, which can lead to, for example, reusing the same `HttpMessageHandler` instance with same injected scoped dependencies between several incoming requests.



Users are strongly advised **not to cache scope-related information** (such as data from `HttpContext`) inside `HttpMessageHandler` instances and use scoped dependencies with caution to avoid leaking sensitive information.

If you require access to an app DI scope from your message handler, for authentication as an example, you'd encapsulate scope-aware logic in a separate transient `DelegatingHandler`, and wrap it around an `HttpMessageHandler` instance from the `IHttpClientFactory` cache. To access the handler call

`IHttpMessageHandlerFactory.CreateHandler` for any registered *named client*. In that case, you'd create an `HttpClient` instance yourself using the constructed handler.



The following example shows creating an `HttpClient` with a scope-aware `DelegatingHandler`:

C#

```

if (scopeAwareHandlerType != null)
{
    if (!typeof(DelegatingHandler).IsAssignableFrom(scopeAwareHandlerType))
    {
        throw new ArgumentException($"""
            Scope aware HttpHandler {scopeAwareHandlerType.Name} should
            be assignable to DelegatingHandler
            """);
    }

    // Create top-most delegating handler with scoped dependencies
    scopeAwareHandler =
    (DelegatingHandler)_scopeServiceProvider.GetRequiredService(scopeAwareHandle
    rType); // should be transient
    if (scopeAwareHandler.InnerHandler != null)
    {
        throw new ArgumentException($"""
            Inner handler of a delegating handler
            {scopeAwareHandlerType.Name} should be null.
            Scope aware HttpHandler should be registered as Transient.
            """);
    }
}

// Get or create HttpResponseMessage from HttpClientFactory
HttpMessageHandler handler = _httpMessageHandlerFactory.CreateHandler(name);

if (scopeAwareHandler != null)

```

```
{  
    scopeAwareHandler.InnerHandler = handler;  
    handler = scopeAwareHandler;  
}  
  
HttpClient client = new(handler);
```

A further workaround can follow with an extension method for registering a scope-aware `DelegatingHandler` and overriding default `IHttpClientFactory` registration by a transient service with access to the current app scope:

C#

```
public static IHttpClientBuilder AddScopeAwareHttpHandler<THandler>(  
    this IHttpClientBuilder builder) where THandler : DelegatingHandler  
{  
    builder.Services.TryAddTransient<THandler>();  
    if (!builder.Services.Any(sd => sd.ImplementationType ==  
typeof(ScopeAwareHttpClientFactory)))  
    {  
        // Override default IHttpClientFactory registration  
        builder.Services.AddTransient<IHttpClientFactory,  
ScopeAwareHttpClientFactory>();  
    }  
  
    builder.Services.Configure<ScopeAwareHttpClientFactoryOptions>(  
        builder.Name, options => options.HttpHandlerType =  
typeof(THandler));  
  
    return builder;  
}
```

For more information, see the [full example ↗](#).

See also

- [Dependency injection in .NET](#)
- [Logging in .NET](#)
- [Configuration in .NET](#)
- [IHttpClientFactory](#)
- [IHttpMessageHandlerFactory](#)
- [HttpClient](#)
- [Make HTTP requests with the HttpClient](#)
- [Implement HTTP retry with exponential backoff](#)

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

.NET is an open source project.
Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

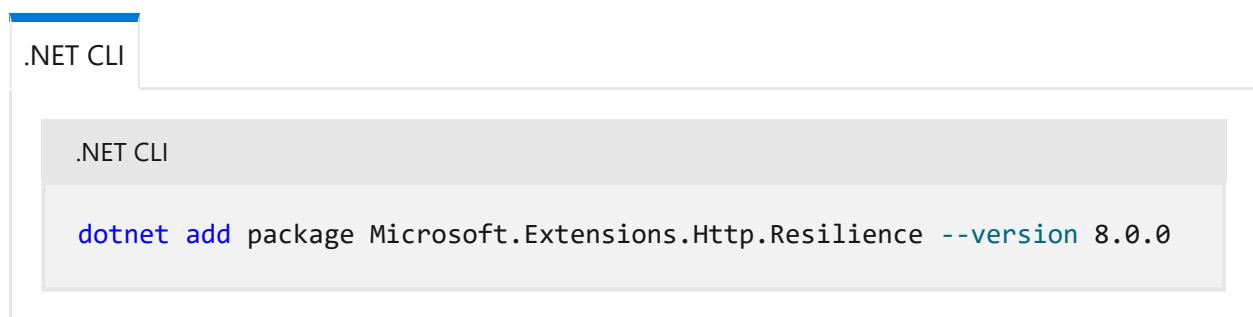
Build resilient HTTP apps: Key development patterns

Article • 10/21/2023

Building robust HTTP apps that can recover from transient fault errors is a common requirement. This article assumes that you've already read [Introduction to resilient app development](#), as this article extends the core concepts conveyed. To help build resilient HTTP apps, the [Microsoft.Extensions.Http.Resilience](#) NuGet package provides resilience mechanisms specifically for the [HttpClient](#). This NuGet package relies on the [Microsoft.Extensions.Resilience](#) library and *Polly*, which is a popular open-source project. For more information, see [Polly](#).

Get started

To use resilience-patterns in HTTP apps, install the [Microsoft.Extensions.Http.Resilience](#) NuGet package.



For more information, see [dotnet add package](#) or [Manage package dependencies in .NET applications](#).

Add resilience to an HTTP client

To add resilience to an [HttpClient](#), you chain a call on the [IHttpClientBuilder](#) type that is returned from calling any of the available [AddHttpClient](#) methods. For more information, see [IHttpClientFactory with .NET](#).

There are several resilience-centric extensions available. Some are standard, thus employing various industry best practices, and others are more customizable. When adding resilience, you should only add one resilience handler and avoid stacking handlers. If you need to add multiple resilience handlers, you should consider using the [AddResilienceHandler](#) extension method, which allows you to customize the resilience strategies.

ⓘ Important

All of the examples within this article rely on the [AddHttpClient](#) API, from the [Microsoft.Extensions.Http](#) library, which returns an [IHttpClientBuilder](#) instance. The [IHttpClientBuilder](#) instance is used to configure the [HttpClient](#) and add the resilience handler.

Add standard resilience handler

The standard resilience handler uses multiple resilience strategies stacked atop one another, with default options to send the requests and handle any transient errors. The standard resilience handler is added by calling the [AddStandardResilienceHandler](#) extension method on an [IHttpClientBuilder](#) instance.

C#

```
var services = new ServiceCollection();

var httpClientBuilder = services.AddHttpClient<ExampleClient>(
    configureClient: static client =>
{
    client.BaseAddress = new("https://jsonplaceholder.typicode.com");
});
```

The preceding code:

- Creates a [ServiceCollection](#) instance.
- Adds an [HttpClient](#) for the [ExampleClient](#) type to the service container.
- Configures the [HttpClient](#) to use `"https://jsonplaceholder.typicode.com"` as the base address.
- Creates the [httpClientBuilder](#) that's used throughout the other examples within this article.

A more real-world example would rely on hosting, such as that described in the [.NET Generic Host](#) article. Using the [Microsoft.Extensions.Hosting](#) NuGet package, consider the following updated example:

C#

```
using Http.Resilience.Example;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
```

```
HostApplicationBuilder builder = Host.CreateApplicationBuilder(args);

IHttpClientBuilder httpClientBuilder =
builder.Services.AddHttpClient<ExampleClient>(
    configureClient: static client =>
{
    client.BaseAddress = new("https://jsonplaceholder.typicode.com");
});
```

The preceding code is similar to the manual `ServiceCollection` creation approach, but instead relies on the `Host.CreateApplicationBuilder()` to build out a host that exposes the services.

The `ExampleClient` is defined as follows:

C#

```
using System.Net.Http.Json;

namespace Http.Resilience.Example;

/// <summary>
/// An example client service, that relies on the <see cref="HttpClient"/>
/// instance.
/// </summary>
/// <param name="client">The given <see cref="HttpClient"/> instance.
/// </param>
internal sealed class ExampleClient(HttpClient client)
{
    /// <summary>
    /// Returns an <see cref="IAsyncEnumerable{T}"/> of <see
    cref="Comment"/>s.
    /// </summary>
    public IAsyncEnumerable<Comment?> GetCommentsAsync()
    {
        return client.GetFromJsonAsAsyncEnumerable<Comment>("/comments");
    }
}
```

The preceding code:

- Defines an `ExampleClient` type that has a constructor that accepts an `HttpClient`.
- Exposes a `GetCommentsAsync` method that sends a GET request to the `/comments` endpoint and returns the response.

The `Comment` type is defined as follows:

C#

```
namespace Http.Resilience.Example;

public record class Comment(
    int PostId, int Id, string Name, string Email, string Body);
```

Given that you've created an [IHttpClientBuilder](#) (`httpClientBuilder`), and you now understand the `ExampleClient` implementation and corresponding `Comment` model, consider the following example:

C#

```
httpClientBuilder.AddStandardResilienceHandler();
```

The preceding code adds the standard resilience handler to the `HttpClient`. Like most resilience APIs, there are overloads that allow you to customize the default options and applied resilience strategies.

Standard resilience handler defaults

The default configuration chains five resilience strategies in the following order (from the outermost to the innermost):

| Order | Strategy | Description |
|-------|-----------------------|---|
| 1 | Rate limiter | The rate limiter pipeline limits the maximum number of concurrent requests being sent to the dependency. |
| 2 | Total request timeout | The total request timeout pipeline applies an overall timeout to the execution, ensuring that the request, including retry attempts, doesn't exceed the configured limit. |
| 3 | Retry | The retry pipeline retries the request in case the dependency is slow or returns a transient error. |
| 4 | Circuit breaker | The circuit breaker blocks the execution if too many direct failures or timeouts are detected. |
| 5 | Attempt timeout | The attempt timeout pipeline limits each request attempt duration and throws if it's exceeded. |

Add standard hedging handler

The standard hedging handler wraps the execution of the request with a standard hedging mechanism. Hedging retries slow requests in parallel.

To use the standard hedging handler, call `AddStandardHedgingHandler` extension method.

The following example configures the `ExampleClient` to use the standard hedging handler.

C#

```
httpClientBuilder.AddStandardHedgingHandler();
```

The preceding code adds the standard hedging handler to the [HttpClient](#).

Standard hedging handler defaults

The standard hedging uses a pool of circuit breakers to ensure that unhealthy endpoints aren't hedged against. By default, the selection from the pool is based on the URL authority (scheme + host + port).

Tip

It's recommended that you configure the way the strategies are selected by calling `StandardHedgingHandlerBuilderExtensions.SelectPipelineByAuthority` or `StandardHedgingHandlerBuilderExtensions.SelectPipelineBy` for more advanced scenarios.

The preceding code adds the standard hedging handler to the [IHttpClientBuilder](#). The default configuration chains five resilience strategies in the following order (from the outermost to the innermost):

| Order | Strategy | Description |
|-------|--------------------------------|--|
| 1 | Total request timeout | The total request timeout pipeline applies an overall timeout to the execution, ensuring that the request, including hedging attempts, doesn't exceed the configured limit. |
| 2 | Hedging | The hedging strategy executes the requests against multiple endpoints in case the dependency is slow or returns a transient error. Routing is options, by default it just hedges the URL provided by the original HttpRequestMessage . |
| 3 | Rate limiter (per endpoint) | The rate limiter pipeline limits the maximum number of concurrent requests being sent to the dependency. |
| 4 | Circuit breaker (per endpoint) | The circuit breaker blocks the execution if too many direct failures or timeouts are detected. |

| Order | Strategy | Description |
|-------|--------------------------------|--|
| 5 | Attempt timeout (per endpoint) | The attempt timeout pipeline limits each request attempt duration and throws if it's exceeded. |

Customize hedging handler route selection

When using the standard hedging handler, you can customize the way the request endpoints are selected by calling various extensions on the `IRoutingStrategyBuilder` type. This can be useful for scenarios such as A/B testing, where you want to route a percentage of the requests to a different endpoint:

C#

```
httpClientBuilder.AddStandardHedgingHandler(static IRoutingStrategyBuilder
builder) =>
{
    // Hedging allows sending multiple concurrent requests
    builder.ConfigureOrderedGroups(static options =>
    {
        options.Groups.Add(new UriEndpointGroup()
        {
            Endpoints =
            {
                // Imagine a scenario where 3% of the requests are
                // sent to the experimental endpoint.
                new() { Uri = new("https://example.net/api/experimental"),
Weight = 3 },
                new() { Uri = new("https://example.net/api/stable"), Weight
= 97 }
            }
        });
    });
}
```

The preceding code:

- Adds the hedging handler to the `IHttpClientBuilder`.
- Configures the `IRoutingStrategyBuilder` to use the `ConfigureOrderedGroups` method to configure the ordered groups.
- Adds an `EndpointGroup` to the `orderedGroup` that routes 3% of the requests to the `https://example.net/api/experimental` endpoint and 97% of the requests to the `https://example.net/api/stable` endpoint.
- Configures the `IRoutingStrategyBuilder` to use the `ConfigureWeightedGroups` method to configure the

To configure a weighted group, call the `ConfigureWeightedGroups` method on the `IRoutingStrategyBuilder` type. The following example configures the `IRoutingStrategyBuilder` to use the `ConfigureWeightedGroups` method to configure the weighted groups.

C#

```
httpClientBuilder.AddStandardHedgingHandler(static IRoutingStrategyBuilder builder) =>
{
    // Hedging allows sending multiple concurrent requests
    builder.ConfigureWeightedGroups(static options =>
    {
        options.SelectionMode = WeightedGroupSelectionMode.EveryAttempt;

        options.Groups.Add(new WeightedUriEndpointGroup()
        {
            Endpoints =
            {
                // Imagine A/B testing
                new() { Uri = new("https://example.net/api/a"), Weight = 33 },
                new() { Uri = new("https://example.net/api/b"), Weight = 33 },
                new() { Uri = new("https://example.net/api/c"), Weight = 33 }
            }
        });
    });
}
```

The preceding code:

- Adds the hedging handler to the `IHttpClientBuilder`.
- Configures the `IRoutingStrategyBuilder` to use the `ConfigureWeightedGroups` method to configure the weighted groups.
- Sets the `SelectionMode` to `WeightedGroupSelectionMode.EveryAttempt`.
- Adds a `WeightedEndpointGroup` to the `weightedGroup` that routes 33% of the requests to the `https://example.net/api/a` endpoint, 33% of the requests to the `https://example.net/api/b` endpoint, and 33% of the requests to the `https://example.net/api/c` endpoint.

💡 Tip

The maximum number of hedging attempts directly correlates to the number of configured groups. For example, if you have two groups, the maximum number of

attempts is two.

For more information, see [Polly docs: Hedging resilience strategy](#).

It's common to configure either an ordered group or weighted group, but it's valid to configure both. Using ordered and weighted groups is helpful in scenarios where you want to send a percentage of the requests to a different endpoint, such is the case with A/B testing.

Add custom resilience handlers

To have more control, you can customize the resilience handlers by using the `AddResilienceHandler` API. This method accepts a delegate that configures the `ResiliencePipelineBuilder<HttpResponseMessage>` instance that is used to create the resilience strategies.

To configure a named resilience handler, call the `AddResilienceHandler` extension method with the name of the handler. The following example configures a named resilience handler called `"CustomPipeline"`.

C#

```
httpClientBuilder.AddResilienceHandler(
    "CustomPipeline",
    static builder =>
{
    // See: https://www.pollydocs.org:strategies/retry.html
    builder.AddRetry(new HttpRetryStrategyOptions
    {
        // Customize and configure the retry logic.
        BackoffType = DelayBackoffType.Exponential,
        MaxRetryAttempts = 5,
        UseJitter = true
    });

    // See: https://www.pollydocs.org:strategies/circuit-breaker.html
    builder.AddCircuitBreaker(new HttpCircuitBreakerStrategyOptions
    {
        // Customize and configure the circuit breaker logic.
        SamplingDuration = TimeSpan.FromSeconds(10),
        FailureRatio = 0.2,
        MinimumThroughput = 3,
        ShouldHandle = static args =>
        {
            return ValueTask.FromResult(args is
            {
                Outcome.Result.StatusCode:
            }
        }
    });
});
```

```

        HttpStatusCode.RequestTimeout or
        HttpStatusCode.TooManyRequests
    );
}
});

// See: https://www.pollydocs.org/strategies/timeout.html
builder.AddTimeout(TimeSpan.FromSeconds(5));
});

```

The preceding code:

- Adds a resilience handler with the name `"CustomPipeline"` as the `pipelineName` to the service container.
- Adds a retry strategy with exponential backoff, five retries, and jitter preference to the resilience builder.
- Adds a circuit breaker strategy with a sampling duration of 10 seconds, a failure ratio of 0.2 (20%), a minimum throughput of three, and a predicate that handles `RequestTimeout` and `TooManyRequests` HTTP status codes to the resilience builder.
- Adds a timeout strategy with a timeout of five seconds to the resilience builder.

There are many options available for each of the resilience strategies. For more information, see the [Polly docs: Strategies](#). For more information about configuring `ShouldHandle` delegates, see [Polly docs: Fault handling in reactive strategies](#).

Dynamic reload

Polly supports dynamic reloading of the configured resilience strategies. This means that you can change the configuration of the resilience strategies at run time. To enable dynamic reload, use the appropriate `AddResilienceHandler` overload that exposes the `ResilienceHandlerContext`. Given the context, call `EnableReloads` of the corresponding resilience strategy options:

C#

```

httpClientBuilder.AddResilienceHandler(
    "AdvancedPipeline",
    static (ResiliencePipelineBuilder<HttpResponseMessage> builder,
        ResilienceHandlerContext context) =>
{
    // Enable reloads whenever the named options change
    context.EnableReloads<HttpRetryStrategyOptions>("RetryOptions");

    // Retrieve the named options
    var retryOptions =
        context.GetOptions<HttpRetryStrategyOptions>("RetryOptions");
}
);

```

```
// Add retries using the resolved options
builder.AddRetry(retryOptions);
});
```

The preceding code:

- Adds a resilience handler with the name `"AdvancedPipeline"` as the `pipelineName` to the service container.
- Enables the reloads of the `"AdvancedPipeline"` pipeline whenever the named `RetryStrategyOptions` options change.
- Retrieves the named options from the `IOptionsMonitor<TOptions>` service.
- Adds a retry strategy with the retrieved options to the resilience builder.

For more information, see [Polly docs: Advanced dependency injection ↗](#).

This example relies on an options section that is capable of change, such as an `appsettings.json` file. Consider the following `appsettings.json` file:

JSON

```
{
  "RetryOptions": {
    "Retry": {
      "Backoff": "Linear",
      "UseJitter": false,
      "MaxRetryAttempts": 7
    }
  }
}
```

Now imagine that these options were bound to the app's configuration, binding the `HttpRetryStrategyOptions` to the `"RetryOptions"` section:

C#

```
var section = builder.Configuration.GetSection("RetryOptions");

builder.Services.Configure<HttpRetryStrategyOptions>(section);
```

For more information, see [Options pattern in .NET](#).

Example usage

Your app relies on [dependency injection](#) to resolve the `ExampleClient` and its corresponding `HttpClient`. The code builds the `IServiceProvider` and resolves the `ExampleClient` from it.

C#

```
IHost host = builder.Build();

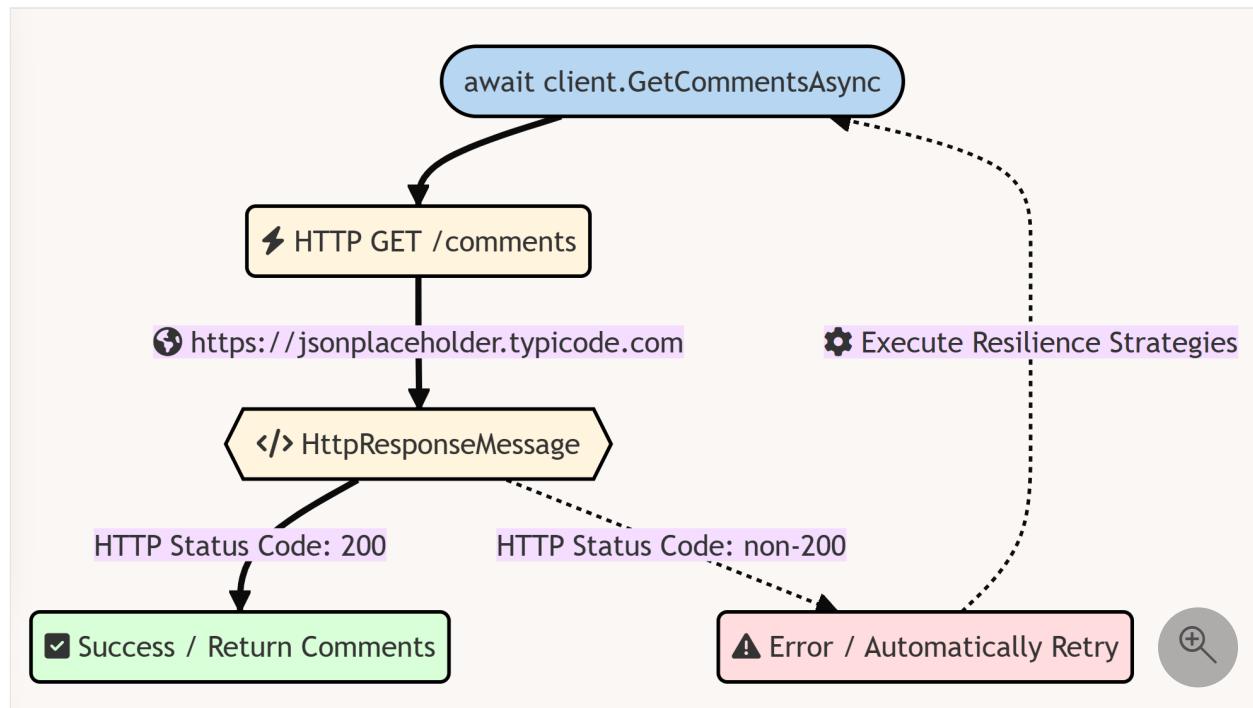
ExampleClient client = host.Services.GetRequiredService<ExampleClient>();

await foreach (Comment? comment in client.GetCommentsAsync())
{
    Console.WriteLine(comment);
}
```

The preceding code:

- Builds the `IServiceProvider` from the `ServiceCollection`.
- Resolves the `ExampleClient` from the `IServiceProvider`.
- Calls the `GetCommentsAsync` method on the `ExampleClient` to get the comments.
- Writes each comment to the console.

Imagine a situation where the network goes down or the server becomes unresponsive. The following diagram shows how the resilience strategies would handle the situation, given the `ExampleClient` and the `GetCommentsAsync` method:



The preceding diagram depicts:

- The `ExampleClient` sends an HTTP GET request to the `/comments` endpoint.

- The [HttpResponseMessage](#) is evaluated:
 - If the response is successful (HTTP 200), the response is returned.
 - If the response is unsuccessful (HTTP non-200), the resilience pipeline employs the configured resilience strategies.

While this is a simple example, it demonstrates how the resilience strategies can be used to handle transient errors. For more information, see [Polly docs: Strategies](#).

 **Collaborate with us on GitHub**

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

 .NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

Use HTTP/3 with HttpClient

Article • 11/17/2023

[HTTP/3](#) is the third and recently standardized major version of HTTP. HTTP/3 uses the same semantics as HTTP/1.1 and HTTP/2: the same request methods, status codes, and message fields apply to all versions. The differences are in the underlying transport. Both HTTP/1.1 and HTTP/2 use TCP as their transport. HTTP/3 uses a transport technology developed alongside HTTP/3 called [QUIC](#).

HTTP/3 and QUIC both have several benefits compared to HTTP/1.1 and HTTP/2:

- Faster response time for the first request. QUIC and HTTP/3 negotiate the connection in fewer round trips between the client and the server. The first request reaches the server faster.
- Improved experience when there's connection packet loss. HTTP/2 multiplexes multiple requests via one TCP connection. Packet loss on the connection affects all requests. This problem is called "head-of-line blocking". Because QUIC provides native multiplexing, lost packets only affect the requests where data has been lost.
- Supports transitioning between networks. This feature is useful for mobile devices where it's common to switch between WIFI and cellular networks as a mobile device changes location. Currently, HTTP/1.1 and HTTP/2 connections fail with an error when switching networks. An app or web browser must retry any failed HTTP requests. HTTP/3 allows the app or web browser to seamlessly continue when a network changes. [HttpClient](#) and Kestrel don't support network transitions in .NET 7. It may be available in a future release.

ⓘ Important

Apps configured to take advantage of HTTP/3 should be designed to also support HTTP/1.1 and HTTP/2. If issues are identified in HTTP/3, it's recommend disabling HTTP/3 until the issues are resolved in a future release of .NET.

HttpClient settings

The HTTP version can be configured by setting `HttpRequestMessage.Version` to 3.0. However, because not all routers, firewalls, and proxies properly support HTTP/3, it's recommend configuring HTTP/3 together with HTTP/1.1 and HTTP/2. In `HttpClient`, this can be done by specifying:

- `HttpRequestMessage.Version` to 1.1.

- `HttpRequestMessage.VersionPolicy` to `HttpVersionPolicy.RequestVersionOrHigher`.

Platform dependencies

HTTP/3 uses QUIC as its transport protocol. The .NET implementation of HTTP/3 uses [MsQuic](#) to provide QUIC functionality. As a result, .NET support of HTTP/3 depends on MsQuic platform requirements. For more information on how to install [MsQuic](#), see [QUIC Platform dependencies](#). If the platform that `HttpClient` is running on doesn't have all the requirements for HTTP/3, then it's disabled.

Using `HttpClient`

The following code example uses [top-level statements](#) and demonstrates how to specify HTTP3 in the request:

C#

```
// See https://aka.ms/new-console-template for more information
using System.Net;

using var client = new HttpClient
{
    DefaultRequestVersion = HttpVersion.Version30,
    DefaultVersionPolicy = HttpVersionPolicy.RequestVersionExact
};

Console.WriteLine("--- localhost:5001 ---");

HttpResponseMessage resp = await client.GetAsync("https://localhost:5001/");
string body = await resp.Content.ReadAsStringAsync();

Console.WriteLine(
    $"status: {resp.StatusCode}, version: {resp.Version}, " +
    $"body: {body.Substring(0, Math.Min(100, body.Length))}");
```

HTTP/3 Support in .NET 6

In .NET 6, HTTP/3 is available as a *preview feature* because the HTTP/3 specification wasn't yet finalized. Behavioral or performance problems may exist in HTTP/3 with .NET 6. For more information on preview features, see [the preview features specification](#).

To enable HTTP/3 support in .NET 6, include the `RuntimeHostConfigurationOption` node in the project file to enable HTTP/3 with `HttpClient`:

XML

```
<ItemGroup>
  <RuntimeHostConfigurationOption Value="true"
    Include="System.Net.SocketsHttpHandler.Http3Support" />
</ItemGroup>
```

Alternatively, you can call [System.AppContext.SetSwitch](#) from your app code, or set the `DOTNET_SYSTEM_NET_HTTP_SOCKETSHTPHANDLER_HTTP3SUPPORT` environment variable to `true`. For more information, see [.NET environment variables](#): `DOTNET_SYSTEM_NET_HTTP_*`.

The reason for requiring a configuration flag for HTTP/3 is to protect apps from future breakage when using version policy `RequestVersionOrHigher`. When calling a server that currently uses HTTP/1.1 and HTTP/2, if the server later upgrades to HTTP/3, the client would try to use HTTP/3 and potentially be incompatible as the standard isn't final and therefore may change after .NET 6 is released.

.NET 6 is only compatible with the 1.9.x versions of libmsquic. Libmsquic 2.x isn't compatible with .NET 6 due to breaking changes in the library. Libmsquic receives updates to 1.9.x when needed to incorporate security fixes.

HTTP/3 Server

HTTP/3 is supported by ASP.NET with the Kestrel server in .NET 6 (as a preview) and .NET 7 (is fully supported). For more information, see [use HTTP/3 with the ASP.NET Core Kestrel web server](#).

Public test servers

Cloudflare hosts a site for HTTP/3 that can be used to test the client at <https://cloudflare-quic.com>.

See also

- [HttpClient](#)
- [HTTP/3 support in Kestrel](#)
- [QUIC support in .NET](#)

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

.NET is an open source project.
Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

Rate limit an HTTP handler in .NET

Article • 03/18/2023

In this article, you'll learn how to create a client-side HTTP handler that rate limits the number of requests it sends. You'll see an [HttpClient](#) that accesses the `"www.example.com"` resource. Resources are consumed by apps that rely on them, and when an app makes too many requests for a single resource, it can lead to *resource contention*. Resource contention occurs when a resource is consumed by too many apps, and the resource is unable to serve all of the apps that are requesting it. This can result in a poor user experience, and in some cases, it can even lead to a denial of service (DoS) attack. For more information on DoS, see [OWASP: Denial of Service ↗](#).

What is rate limiting?

Rate limiting is the concept of limiting how much a resource can be accessed. For example, you may know that a database your app accesses can safely handle 1,000 requests per minute, but it may not handle much more than that. You can put a rate limiter in your app that only allows 1,000 requests every minute and rejects any more requests before they can access the database. Thus, rate limiting your database and allowing your app to handle a safe number of requests. This is a common pattern in distributed systems, where you may have multiple instances of an app running, and you want to ensure that they don't all try to access the database at the same time. There are multiple different rate-limiting algorithms to control the flow of requests.

To use rate limiting in .NET, you'll reference the [System.Threading.RateLimiting ↗](#) NuGet package.

Implement a `DelegatingHandler` subclass

To control the flow of requests, you implement a custom [DelegatingHandler](#) subclass. This is a type of [HttpMessageHandler](#) that allows you to intercept and handle requests before they're sent to the server. You can also intercept and handle responses before they're returned to the caller. In this example, you'll implement a custom `DelegatingHandler` subclass that limits the number of requests that can be sent to a single resource. Consider the following custom `ClientSideRateLimitedHandler` class:

C#

```
internal sealed class ClientSideRateLimitedHandler  
    : DelegatingHandler, IAsyncDisposable
```

```

{
    private readonly RateLimiter _rateLimiter;

    public ClientSideRateLimitedHandler(RateLimiter limiter)
        : base(new HttpClientHandler()) => _rateLimiter = limiter;

    protected override async Task<HttpResponseMessage> SendAsync(
        HttpRequestMessage request, CancellationToken cancellationToken)
    {
        using RateLimitLease lease = await _rateLimiter.AcquireAsync(
            permitCount: 1, cancellationToken);

        if (lease.IsAcquired)
        {
            return await base.SendAsync(request, cancellationToken);
        }

        var response = new
        HttpResponseMessage(HttpStatusCode.TooManyRequests);
        if (lease.TryGetMetadata(
            MetadataName.RetryAfter, out TimeSpan retryAfter))
        {
            response.Headers.Add(
                "Retry-After",
                ((int)retryAfter.TotalSeconds).ToString(
                    NumberFormatInfo.InvariantInfo));
        }

        return response;
    }

    async ValueTask IAsyncDisposable.DisposeAsync()
    {
        await _rateLimiter.DisposeAsync().ConfigureAwait(false);

        Dispose(disposing: false);
        GC.SuppressFinalize(this);
    }
}

protected override void Dispose(bool disposing)
{
    base.Dispose(disposing);

    if (disposing)
    {
        _rateLimiter.Dispose();
    }
}
}

```

The preceding C# code:

- Inherits the `DelegatingHandler` type.

- Implements the [IAsyncDisposable](#) interface.
- Defines a `RateLimiter` field that is assigned from the constructor.
- Overrides the `SendAsync` method to intercept and handle requests before they're sent to the server.
- Overrides the `DisposeAsync()` method to dispose of the `RateLimiter` instance.

Looking a bit closer at the `SendAsync` method, you'll see that it:

- Relies on the `RateLimiter` instance to acquire a `RateLimitLease` from the `AcquireAsync`.
- When the `lease.IsAcquired` property is `true`, the request is sent to the server.
- Otherwise, an [HttpResponseMessage](#) is returned with a `429` status code, and if the `lease` contains a `RetryAfter` value, the `Retry-After` header is set to that value.

Emulate many concurrent requests

To put this custom `DelegatingHandler` subclass to the test, you'll create a console app that emulates many concurrent requests. This `Program` class creates an [HttpClient](#) with the custom `ClientSideRateLimitedHandler`:

C#

```
var options = new TokenBucketRateLimiterOptions
{
    TokenLimit = 8,
    QueueProcessingOrder = QueueProcessingOrder.OldestFirst,
    QueueLimit = 3,
    ReplenishmentPeriod = TimeSpan.FromMilliseconds(1),
    TokensPerPeriod = 2,
    AutoReplenishment = true
};

// Create an HTTP client with the client-side rate limited handler.
using HttpClient client = new(
    handler: new ClientSideRateLimitedHandler(
        limiter: new TokenBucketRateLimiter(options)));

// Create 100 urls with a unique query string.
var oneHundredUrls = Enumerable.Range(0, 100).Select(
    i => $"https://example.com?iteration={i:0#}");

// Flood the HTTP client with requests.
var floodOneThroughFortyNineTask = Parallel.ForEachAsync(
    source: oneHundredUrls.Take(0..49),
    body: (url, cancellationToken) => GetAsync(client, url,
    cancellationToken));
```

```

var floodFiftyThroughOneHundredTask = Parallel.ForEachAsync(
    source: oneHundredUrls.Take(^50..),
    body: (url, cancellationToken) => GetAsync(client, url,
    cancellationToken));

await Task.WhenAll(
    floodOneThroughFortyNineTask,
    floodFiftyThroughOneHundredTask);

static async ValueTask GetAsync(
    HttpClient client, string url, CancellationToken cancellationToken)
{
    using var response =
        await client.GetAsync(url, cancellationToken);

    Console.WriteLine(
        $"URL: {url}, HTTP status code: {response.StatusCode}
({(int)response.StatusCode})");
}

```

In the preceding console app:

- The `TokenBucketRateLimiterOptions` are configured with a token limit of `8`, and queue processing order of `oldestFirst`, a queue limit of `3`, and replenishment period of `1` millisecond, a tokens per period value of `2`, and an auto-replenish value of `true`.
- An `HttpClient` is created with the `ClientSideRateLimitedHandler` that is configured with the `TokenBucketRateLimiter`.
- To emulate 100 requests, `Enumerable.Range` creates 100 URLs, each with a unique query string parameter.
- Two `Task` objects are assigned from the `Parallel.ForEachAsync` method, splitting the URLs into two groups.
- The `HttpClient` is used to send a `GET` request to each URL, and the response is written to the console.
- `Task.WhenAll` waits for both tasks to complete.

Since the `HttpClient` is configured with the `ClientSideRateLimitedHandler`, not all requests will make it to the server resource. You can test this assertion by running the console app. You'll see that only a fraction of the total number of requests are sent to the server, and the rest are rejected with an HTTP status code of `429`. Try altering the `options` object used to create the `TokenBucketRateLimiter` to see how the number of requests that are sent to the server changes.

Consider the following example output:

Output

URL: https://example.com?iteration=06, HTTP status code: TooManyRequests (429)
URL: https://example.com?iteration=60, HTTP status code: TooManyRequests (429)
URL: https://example.com?iteration=55, HTTP status code: TooManyRequests (429)
URL: https://example.com?iteration=59, HTTP status code: TooManyRequests (429)
URL: https://example.com?iteration=57, HTTP status code: TooManyRequests (429)
URL: https://example.com?iteration=11, HTTP status code: TooManyRequests (429)
URL: https://example.com?iteration=63, HTTP status code: TooManyRequests (429)
URL: https://example.com?iteration=13, HTTP status code: TooManyRequests (429)
URL: https://example.com?iteration=62, HTTP status code: TooManyRequests (429)
URL: https://example.com?iteration=65, HTTP status code: TooManyRequests (429)
URL: https://example.com?iteration=64, HTTP status code: TooManyRequests (429)
URL: https://example.com?iteration=67, HTTP status code: TooManyRequests (429)
URL: https://example.com?iteration=14, HTTP status code: TooManyRequests (429)
URL: https://example.com?iteration=68, HTTP status code: TooManyRequests (429)
URL: https://example.com?iteration=16, HTTP status code: TooManyRequests (429)
URL: https://example.com?iteration=69, HTTP status code: TooManyRequests (429)
URL: https://example.com?iteration=70, HTTP status code: TooManyRequests (429)
URL: https://example.com?iteration=71, HTTP status code: TooManyRequests (429)
URL: https://example.com?iteration=17, HTTP status code: TooManyRequests (429)
URL: https://example.com?iteration=18, HTTP status code: TooManyRequests (429)
URL: https://example.com?iteration=72, HTTP status code: TooManyRequests (429)
URL: https://example.com?iteration=73, HTTP status code: TooManyRequests (429)
URL: https://example.com?iteration=74, HTTP status code: TooManyRequests (429)
URL: https://example.com?iteration=19, HTTP status code: TooManyRequests (429)
URL: https://example.com?iteration=75, HTTP status code: TooManyRequests (429)
URL: https://example.com?iteration=76, HTTP status code: TooManyRequests (429)
URL: https://example.com?iteration=79, HTTP status code: TooManyRequests (429)

```
URL: https://example.com?iteration=77, HTTP status code: TooManyRequests  
(429)  
URL: https://example.com?iteration=21, HTTP status code: TooManyRequests  
(429)  
URL: https://example.com?iteration=78, HTTP status code: TooManyRequests  
(429)  
URL: https://example.com?iteration=81, HTTP status code: TooManyRequests  
(429)  
URL: https://example.com?iteration=22, HTTP status code: TooManyRequests  
(429)  
URL: https://example.com?iteration=80, HTTP status code: TooManyRequests  
(429)  
URL: https://example.com?iteration=20, HTTP status code: TooManyRequests  
(429)  
URL: https://example.com?iteration=82, HTTP status code: TooManyRequests  
(429)  
URL: https://example.com?iteration=83, HTTP status code: TooManyRequests  
(429)  
URL: https://example.com?iteration=23, HTTP status code: TooManyRequests  
(429)  
URL: https://example.com?iteration=84, HTTP status code: TooManyRequests  
(429)  
URL: https://example.com?iteration=24, HTTP status code: TooManyRequests  
(429)  
URL: https://example.com?iteration=85, HTTP status code: TooManyRequests  
(429)  
URL: https://example.com?iteration=86, HTTP status code: TooManyRequests  
(429)  
URL: https://example.com?iteration=25, HTTP status code: TooManyRequests  
(429)  
URL: https://example.com?iteration=87, HTTP status code: TooManyRequests  
(429)  
URL: https://example.com?iteration=26, HTTP status code: TooManyRequests  
(429)  
URL: https://example.com?iteration=88, HTTP status code: TooManyRequests  
(429)  
URL: https://example.com?iteration=89, HTTP status code: TooManyRequests  
(429)  
URL: https://example.com?iteration=27, HTTP status code: TooManyRequests  
(429)  
URL: https://example.com?iteration=90, HTTP status code: TooManyRequests  
(429)  
URL: https://example.com?iteration=28, HTTP status code: TooManyRequests  
(429)  
URL: https://example.com?iteration=91, HTTP status code: TooManyRequests  
(429)  
URL: https://example.com?iteration=94, HTTP status code: TooManyRequests  
(429)  
URL: https://example.com?iteration=29, HTTP status code: TooManyRequests  
(429)  
URL: https://example.com?iteration=93, HTTP status code: TooManyRequests  
(429)  
URL: https://example.com?iteration=96, HTTP status code: TooManyRequests  
(429)  
URL: https://example.com?iteration=92, HTTP status code: TooManyRequests
```

(429)
URL: https://example.com?iteration=95, HTTP status code: TooManyRequests
(429)
URL: https://example.com?iteration=31, HTTP status code: TooManyRequests
(429)
URL: https://example.com?iteration=30, HTTP status code: TooManyRequests
(429)
URL: https://example.com?iteration=97, HTTP status code: TooManyRequests
(429)
URL: https://example.com?iteration=98, HTTP status code: TooManyRequests
(429)
URL: https://example.com?iteration=99, HTTP status code: TooManyRequests
(429)
URL: https://example.com?iteration=32, HTTP status code: TooManyRequests
(429)
URL: https://example.com?iteration=33, HTTP status code: TooManyRequests
(429)
URL: https://example.com?iteration=34, HTTP status code: TooManyRequests
(429)
URL: https://example.com?iteration=35, HTTP status code: TooManyRequests
(429)
URL: https://example.com?iteration=36, HTTP status code: TooManyRequests
(429)
URL: https://example.com?iteration=37, HTTP status code: TooManyRequests
(429)
URL: https://example.com?iteration=38, HTTP status code: TooManyRequests
(429)
URL: https://example.com?iteration=39, HTTP status code: TooManyRequests
(429)
URL: https://example.com?iteration=40, HTTP status code: TooManyRequests
(429)
URL: https://example.com?iteration=41, HTTP status code: TooManyRequests
(429)
URL: https://example.com?iteration=42, HTTP status code: TooManyRequests
(429)
URL: https://example.com?iteration=43, HTTP status code: TooManyRequests
(429)
URL: https://example.com?iteration=44, HTTP status code: TooManyRequests
(429)
URL: https://example.com?iteration=45, HTTP status code: TooManyRequests
(429)
URL: https://example.com?iteration=46, HTTP status code: TooManyRequests
(429)
URL: https://example.com?iteration=47, HTTP status code: TooManyRequests
(429)
URL: https://example.com?iteration=48, HTTP status code: TooManyRequests
(429)
URL: https://example.com?iteration=15, HTTP status code: OK (200)
URL: https://example.com?iteration=04, HTTP status code: OK (200)
URL: https://example.com?iteration=54, HTTP status code: OK (200)
URL: https://example.com?iteration=08, HTTP status code: OK (200)
URL: https://example.com?iteration=00, HTTP status code: OK (200)
URL: https://example.com?iteration=51, HTTP status code: OK (200)
URL: https://example.com?iteration=10, HTTP status code: OK (200)
URL: https://example.com?iteration=66, HTTP status code: OK (200)

```
URL: https://example.com?iteration=56, HTTP status code: OK (200)
URL: https://example.com?iteration=52, HTTP status code: OK (200)
URL: https://example.com?iteration=12, HTTP status code: OK (200)
URL: https://example.com?iteration=53, HTTP status code: OK (200)
URL: https://example.com?iteration=07, HTTP status code: OK (200)
URL: https://example.com?iteration=02, HTTP status code: OK (200)
URL: https://example.com?iteration=01, HTTP status code: OK (200)
URL: https://example.com?iteration=61, HTTP status code: OK (200)
URL: https://example.com?iteration=05, HTTP status code: OK (200)
URL: https://example.com?iteration=09, HTTP status code: OK (200)
URL: https://example.com?iteration=03, HTTP status code: OK (200)
URL: https://example.com?iteration=58, HTTP status code: OK (200)
URL: https://example.com?iteration=50, HTTP status code: OK (200)
```

You'll notice that the first logged entries are always the immediately returned 429 responses, and the last entries are always the 200 responses. This is because the rate limit is encountered client-side and avoids making an HTTP call to a server. This is a good thing because it means that the server isn't flooded with requests. It also means that the rate limit is enforced consistently across all clients.

Note also that each URL's query string is unique: examine the `iteration` parameter to see that it's incremented by one for each request. This parameter helps to illustrate that the 429 responses aren't from the first requests, but rather from the requests that are made after the rate limit is reached. The 200 responses arrive later but these requests were made earlier—before the limit was reached.

To have a better understanding of the various rate-limiting algorithms, try rewriting this code to accept a different `RateLimiter` implementation. In addition to the `TokenBucketRateLimiter` you could try:

- `ConcurrencyLimiter`
- `FixedWindowRateLimiter`
- `PartitionedRateLimiter`
- `SlidingWindowRateLimiter`

Summary

In this article, you learned how to implement a custom `ClientSideRateLimitedHandler`. This pattern could be used to implement a rate-limited HTTP client for resources that you know have API limits. In this way, you're preventing your client app from making unnecessary requests to the server, and you're also preventing your app from being blocked by the server. Additionally, with the use of metadata to store retry timing values, you could also implement automatic retry logic.

See also

- [Announcing Rate Limiting for .NET ↗](#)
- [Rate limiting middleware in ASP.NET Core](#)
- [Azure Architecture: Rate limiting pattern](#)
- [Automatic retry logic in .NET](#)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

Customize SNI in HTTP requests

Article • 10/17/2023

When a client and server negotiate an HTTPS connection, a TLS connection needs to be established first. As part of the TLS handshake, the client sends the domain name of the server it's connecting to in one of the TLS extensions. When multiple (virtual) servers are hosted on the same machine, this feature of the TLS protocol allows clients to distinguish which of these servers they're connecting to and to configure TLS settings, such as the server certificate, accordingly.

When an HTTP request using `HttpClient` is made, the implementation automatically selects a value for the server name indication (SNI) extension based on the URL the client is connecting to. For scenarios that require more manual control of the extension, you can use one of the following approaches.

Host header

Host HTTP header performs a similar function as the SNI extension in TLS. It lets the target server distinguish among requests for multiple host names on a single IP address. `HttpClient` automatically fills in the Host header using the request URI. However, you can also set its value manually, and `HttpClient` will also use the new value in the SNI extension. You can use either `HttpRequestMessage.Headers.Host` or `HttpClient.DefaultRequestHeaders.Host` to achieve this effect.

C#

```
using HttpClient client = new();

client.DefaultRequestHeaders.Host = "www.microsoft.com";

using var response = await client.GetAsync("https://127.0.0.1:5001/");

System.Console.WriteLine(response);
```

ⓘ Note

This method doesn't allow you to avoid sending SNI altogether when connecting to a URL with a hostname. If the header is set to empty string, `HttpClient` uses the hostname from the URL instead.

⚠ Note

Customizing the Host header affects server certificate validation. By default, client will expect the server certificate to match the hostname in the Host header.

Manual SslStream authentication via ConnectCallback

A more complicated, but also more powerful, option is to use the `SocketsHttpHandler.ConnectCallback`. Since .NET 7, it's possible to return an authenticated `SslStream` and thus customize how the TLS connection is established. Inside the callback, arbitrary `SslClientAuthenticationOptions` options can be used to perform client-side authentication.

C#

```
var handler = new SocketsHttpHandler
{
    ConnectCallback = async (context, cancellationToken) =>
    {
        var socket = new Socket(SocketType.Stream, ProtocolType.Tcp) { NoDelay = true };
        try
        {
            await socket.ConnectAsync(context.DnsEndPoint,
                cancellationToken);

            var sslStream = new SslStream(new NetworkStream(socket,
                ownsSocket: true));

            // When using HTTP/2, you must also keep in mind to set options like ApplicationProtocols
            await sslStream.AuthenticateAsClientAsync(new SslClientAuthenticationOptions
            {
                TargetHost = context.DnsEndPoint.Host,
            }, cancellationToken);

            return sslStream;
        }
        catch
        {
            socket.Dispose();
            throw;
        }
    }
}
```

```
};

using HttpClient client = new(handler);

using var response = await client.GetAsync("https://www.microsoft.com");

System.Console.WriteLine(response);
```

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

System.Net.Http.HttpClient class

Article • 01/08/2024

This article provides supplementary remarks to the reference documentation for this API.

The [HttpClient](#) class instance acts as a session to send HTTP requests. An [HttpClient](#) instance is a collection of settings applied to all requests executed by that instance. In addition, every [HttpClient](#) instance uses its own connection pool, isolating its requests from requests executed by other [HttpClient](#) instances.

Instancing

[HttpClient](#) is intended to be instantiated once and reused throughout the life of an application. In .NET Core and .NET 5+, [HttpClient](#) pools connections inside the handler instance and reuses a connection across multiple requests. If you instantiate an [HttpClient](#) class for every request, the number of sockets available under heavy loads will be exhausted. This exhaustion will result in [SocketException](#) errors.

You can configure additional options by passing in a "handler", such as [HttpClientHandler](#) (or [SocketsHttpHandler](#) in .NET Core 2.1 or later), as part of the constructor. The connection properties on the handler cannot be changed once a request has been submitted, so one reason to create a new [HttpClient](#) instance would be if you need to change the connection properties. If different requests require different settings, this may also lead to an application having multiple [HttpClient](#) instances, where each instance is configured appropriately, and then requests are issued on the relevant client.

[HttpClient](#) only resolves DNS entries when a connection is created. It does not track any time to live (TTL) durations specified by the DNS server. If DNS entries change regularly, which can happen in some container scenarios, the client won't respect those updates. To solve this issue, you can limit the lifetime of the connection by setting the [SocketsHttpHandler.PooledConnectionLifetime](#) property, so that DNS lookup is required when the connection is replaced.

C#

```
public class GoodController : ApiController
{
    private static readonly HttpClient httpClient;

    static GoodController()
```

```

    {
        var socketsHandler = new SocketsHttpHandler
        {
            PooledConnectionLifetime = TimeSpan.FromMinutes(2)
        };

        httpClient = new HttpClient(socketsHandler);
    }
}

```

As an alternative to creating only one `HttpClient` instance, you can also use [IHttpClientFactory](#) to manage the `HttpClient` instances for you. For more information, see [Guidelines for using HttpClient](#).

Derivation

The `HttpClient` also acts as a base class for more specific HTTP clients. An example would be a `FacebookHttpClient` that provides additional methods specific to a Facebook web service (for example, a `GetFriends` method). Derived classes should not override the virtual methods on the class. Instead, use a constructor overload that accepts `HttpMessageHandler` to configure any pre-request or post-request processing.

Transports

The `HttpClient` is a high-level API that wraps the lower-level functionality available on each platform where it runs.

On each platform, `HttpClient` tries to use the best available transport:

[] [Expand table](#)

| Host/Runtime | Backend |
|---------------------------|--|
| Windows/.NET Framework | HttpWebRequest |
| Windows/Mono | HttpWebRequest |
| Windows/UWP | Windows native WinHttpHandler (HTTP 2.0 capable) |
| Windows/.NET Core 1.0-2.0 | Windows native WinHttpHandler (HTTP 2.0 capable) |
| Android/Xamarin | Selected at build-time. Can either use HttpWebRequest or be configured to use Android's native HttpURLConnection ↗ |

| Host/Runtime | Backend |
|-------------------------------|---|
| iOS, tvOS, watchOS/Xamarin | Selected at build-time. Can either use HttpWebRequest or be configured to use Apple's NSURLSession (HTTP 2.0 capable) |
| macOS/Xamarin | Selected at build-time. Can either use HttpWebRequest or be configured to use Apple's NSURLSession (HTTP 2.0 capable) |
| macOS/Mono | HttpWebRequest |
| macOS/.NET Core 1.0- 2.0 | <code>libcurl</code> -based HTTP transport (HTTP 2.0 capable) |
| Linux/Mono | HttpWebRequest |
| Linux/.NET Core 1.0- 2.0 | <code>libcurl</code> -based HTTP transport (HTTP 2.0 capable) |
| .NET Core 2.1 and later | System.Net.Http.SocketsHttpHandler |

Users can also configure a specific transport for [HttpClient](#) by invoking the [HttpClient](#) constructor that takes an [HttpMessageHandler](#).

.NET Framework & Mono

By default on .NET Framework and Mono, [HttpWebRequest](#) is used to send requests to the server. This behavior can be modified by specifying a different handler in one of the constructor overloads with an [HttpMessageHandler](#) parameter. If you require features like authentication or caching, you can use [WebRequestHandler](#) to configure settings and the instance can be passed to the constructor. The returned handler can be passed to a constructor overload that has an [HttpMessageHandler](#) parameter.

.NET Core

Starting with .NET Core 2.1, the [System.Net.Http.SocketsHttpHandler](#) class instead of [HttpClientHandler](#) provides the implementation used by higher-level HTTP networking classes such as [HttpClient](#). The use of [SocketsHttpHandler](#) offers a number of advantages:

- A significant performance improvement when compared with the previous implementation.
- The elimination of platform dependencies, which simplifies deployment and servicing. For example, `libcurl` is no longer a dependency on .NET Core for macOS and .NET Core for Linux.

- Consistent behavior across all .NET platforms.

If this change is undesirable, on Windows you can continue to use [WinHttpHandler](#) by referencing its [NuGet package](#) and passing it to [HttpClient's constructor](#) manually.

Configure behavior using runtime configuration options

Certain aspects of [HttpClient](#)'s behavior are customizable through [Runtime configuration options](#). However, the behavior of these switches differs through .NET versions. For example, in .NET Core 2.1 - 3.1, you can configure whether [SocketsHttpHandler](#) is used by default, but that option is no longer available starting in .NET 5.0.

Connection pooling

[HttpClient](#) pools HTTP connections where possible and uses them for more than one request. This can have a significant performance benefit, especially for HTTPS requests, as the connection handshake is only done once.

Connection pool properties can be configured on a [HttpClientHandler](#) or [SocketsHttpHandler](#) passed in during construction, including [MaxConnectionsPerServer](#), [PooledConnectionIdleTimeout](#), and [PooledConnectionLifetime](#).

Disposing of the [HttpClient](#) instance closes the open connections and cancels any pending requests.

Note

If you concurrently send HTTP/1.1 requests to the same server, new connections can be created. Even if you reuse the `HttpClient` instance, if the rate of requests is high, or if there are any firewall limitations, that can exhaust the available sockets because of default TCP cleanup timers. To limit the number of concurrent connections, you can set the `MaxConnectionsPerServer` property. By default, the number of concurrent HTTP/1.1 connections is unlimited.

Buffering and request lifetime

By default, `HttpClient` methods (except `GetStreamAsync`) buffer the responses from the server, reading all the response body into memory before returning the async result. Those requests will continue until one of the following occurs:

- The `Task<TResult>` succeeds and returns a result.
- The `Timeout` is reached, in which case the `Task<TResult>` will be cancelled.
- The `CancellationToken` passable to some method overloads is fired.
- `CancelPendingRequests()` is called.
- The `HttpClient` is disposed.

You can change the buffering behavior on a per-request basis using the `HttpCompletionOption` parameter available on some method overloads. This argument can be used to specify if the `Task<TResult>` should be considered complete after reading just the response headers, or after reading and buffering the response content.

If your app that uses `HttpClient` and related classes in the `System.Net.Http` namespace intends to download large amounts of data (50 megabytes or more), then the app should stream those downloads and not use the default buffering. If you use the default buffering, the client memory usage will get very large, potentially resulting in substantially reduced performance.

Thread safety

The following methods are thread safe:

- `CancelPendingRequests`
- `DeleteAsync`
- `GetAsync`
- `GetByteArrayAsync`
- `GetStreamAsync`
- `GetStringAsync`
- `PostAsync`
- `PutAsync`
- `SendAsync`

Proxies

By default, `HttpClient` reads proxy configuration from environment variables or user/system settings, depending on the platform. You can change this behavior by passing a `WebProxy` or `IWebProxy` to, in order of precedence:

- The [Proxy](#) property on a `HttpClientHandler` passed in during `HttpClient` construction
- The [DefaultProxy](#) static property (affects all instances)

You can disable the proxy using [UseProxy](#). The default configuration for Windows users is to try and detect a proxy using network discovery, which can be slow. For high throughput applications where it's known that a proxy isn't required, you should disable the proxy.

Proxy settings (like [Credentials](#)) should be changed only before the first request is made using the `HttpClient`. Changes made after using the `HttpClient` for the first time may not be reflected in subsequent requests.

Timeouts

You can use [Timeout](#) to set a default timeout for all HTTP requests from the `HttpClient` instance. The timeout only applies to the `xxxAsync` methods that cause a request/response to be initiated. If the timeout is reached, the [Task<TResult>](#) for that request is cancelled.

You can set some additional timeouts if you pass in a [SocketsHttpHandler](#) instance when constructing the `HttpClient` object:

[\[\]](#) [Expand table](#)

| Property | Description |
|---|--|
| ConnectTimeout | Specifies a timeout that's used when a request requires a new TCP connection to be created. If the timeout occurs, the request Task<TResult> is cancelled. |
| PooledConnectionLifetime | Specifies a timeout to be used for each connection in the connection pool. If the connection is idle, the connection is immediately closed; otherwise, the connection is closed at the end of the current request. |
| PooledConnectionIdleTimeout | If a connection in the connection pool is idle for this long, the connection is closed. |
| Expect100ContinueTimeout | If request has an "Expect: 100-continue" header, it delays sending content until the timeout or until a "100-continue" response is received. |

`HttpClient` only resolves DNS entries when the connections are created. It does not track any time to live (TTL) durations specified by the DNS server. If DNS entries are changing

regularly, which can happen in some container scenarios, you can use the [PooledConnectionLifetime](#) to limit the lifetime of the connection so that DNS lookup is required when replacing the connection.

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

 .NET feedback

.NET is an open source project.
Select a link to provide feedback:

-  [Open a documentation issue](#)
-  [Provide product feedback](#)

System.Net.Http.HttpClientHandler class

Article • 01/08/2024

This article provides supplementary remarks to the reference documentation for this API.

The [HttpClientHandler](#) class and classes derived from it enable developers to configure a variety of options ranging from proxies to authentication.

HttpClientHandler in .NET Core

Starting in .NET Core 2.1, the implementation of the `HttpClientHandler` class was changed to be based on the cross-platform HTTP protocol stack used by the [System.Net.Http.SocketsHttpHandler](#) class. Prior to .NET Core 2.1, the `HttpClientHandler` class used older HTTP protocol stacks ([WinHttpHandler](#) on Windows and `CurlHandler`, an internal class implemented on top of Linux's native `libcurl` component, on Linux).

On .NET Core 2.1 - 3.1 only, you can configure your app to use the older HTTP protocol stacks in one of the following three ways:

- Call the [AppContext.SetSwitch](#) method:

C#

```
AppContext.SetSwitch("System.Net.Http.UseSocketsHttpHandler", false);
```

VB

```
AppContext.SetSwitch("System.Net.Http.UseSocketsHttpHandler", False)
```

- Define the `System.Net.Http.UseSocketsHttpHandler` switch in the `.netcore.runtimeconfig.json` configuration file:

JSON

```
"runtimeOptions": {
    "configProperties": {
        "System.Net.Http.UseSocketsHttpHandler": false
    }
}
```

- Define an environment variable named `DOTNET_SYSTEM_NET_HTTP_USESOCKETSHTTPHANDLER` and set it to either `false` or `0`.

These configuration options are not available starting with .NET 5.

 **Collaborate with us on GitHub**

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

.NET is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

System.Net.HttpListener class

Article • 01/08/2024

This article provides supplementary remarks to the reference documentation for this API.

Using the [HttpListener](#) class, you can create a simple HTTP protocol listener that responds to HTTP requests. The listener is active for the lifetime of the [HttpListener](#) object and runs within your application with its permissions.

To use [HttpListener](#), create a new instance of the class using the [HttpListener](#) constructor and use the [Prefixes](#) property to gain access to the collection that holds the strings that specify which Uniform Resource Identifier (URI) prefixes the [HttpListener](#) should process.

A URI prefix string is composed of a scheme (http or https), a host, an optional port, and an optional path. An example of a complete prefix string is

`http://www.contoso.com:8080/customerData/`. Prefixes must end in a forward slash ("/").

The [HttpListener](#) object with the prefix that most closely matches a requested URI responds to the request. Multiple [HttpListener](#) objects cannot add the same prefix; a [Win32Exception](#) exception is thrown if a [HttpListener](#) adds a prefix that is already in use.

When a port is specified, the host element can be replaced with "*" to indicate that the [HttpListener](#) accepts requests sent to the port if the requested URI does not match any other prefix. For example, to receive all requests sent to port 8080 when the requested URI is not handled by any [HttpListener](#), the prefix is `http://*:8080/`. Similarly, to specify that the [HttpListener](#) accepts all requests sent to a port, replace the host element with the "+" character. For example, `https://+:8080`. The "*" and "+" characters can be present in prefixes that include paths.

Starting with .NET Core 2.0 or .NET Framework 4.6 on Windows 10, wildcard subdomains are supported in URI prefixes that are managed by an [HttpListener](#) object. To specify a wildcard subdomain, use the "*" character as part of the hostname in a URI prefix. For example, `http://*.foo.com/`. Pass this as the argument to the [Add](#) method.

⚠ Warning

Top-level wildcard bindings (`http://*:8080/` and `http://+:8080`) should **not** be used. Top-level wildcard bindings can open up your app to security vulnerabilities. This applies to both strong and weak wildcards. Use explicit host names rather than wildcards. Subdomain wildcard binding (for example, `*.mysub.com`) doesn't have

this security risk if you control the entire parent domain (as opposed to `*.com`, which is vulnerable). See [rfc7230 section-5.4](#) for more information.

To begin listening for requests from clients, add the URI prefixes to the collection and call the [Start](#) method. [HttpListener](#) offers both synchronous and asynchronous models for processing client requests. Requests and their associated responses are accessed using the [HttpListenerContext](#) object returned by the [GetContext](#) method or its asynchronous counterparts, the [BeginGetContext](#) and [EndGetContext](#) methods.

The synchronous model is appropriate if your application should block while waiting for a client request and if you want to process only one request at a time. Using the synchronous model, call the [GetContext](#) method, which waits for a client to send a request. The method returns an [HttpListenerContext](#) object to you for processing when one occurs.

In the more complex asynchronous model, your application does not block while waiting for requests and each request is processed in its own execution thread. Use the [BeginGetContext](#) method to specify an application-defined method to be called for each incoming request. Within that method, call the [EndGetContext](#) method to obtain the request, process it, and respond.

In either model, incoming requests are accessed using the [HttpListenerContext.Request](#) property and are represented by [HttpListenerRequest](#) objects. Similarly, responses are accessed using the [HttpListenerContext.Response](#) property and are represented by [HttpListenerResponse](#) objects. These objects share some functionality with the [HttpWebRequest](#) and [HttpWebResponse](#) objects, but the latter objects cannot be used in conjunction with [HttpListener](#) because they implement client, not server, behaviors.

An [HttpListener](#) can require client authentication. You can either specify a particular scheme to use for authentication, or you can specify a delegate that determines the scheme to use. You must require some form of authentication to obtain information about the client's identity. For additional information, see the [User](#), [AuthenticationSchemes](#), and [AuthenticationSchemeSelectorDelegate](#) properties.

① Note

If you create an [HttpListener](#) using `https`, you must select a Server Certificate for that listener. Otherwise, requests to this [HttpListener](#) will fail with an unexpected close of the connection.

① Note

You can configure Server Certificates and other listener options by using Network Shell (netsh.exe). See [Network Shell \(Netsh\)](#) for more details. The executable began shipping with Windows Server 2008 and Windows Vista.

Note

If you specify multiple authentication schemes for the [HttpListener](#), the listener will challenge clients in the following order: `Negotiate`, `NTLM`, `Digest`, and then `Basic`.

HTTP.sys

The [HttpListener](#) class is built on top of `HTTP.sys`, which is the kernel mode listener that handles all HTTP traffic for Windows. `HTTP.sys` provides connection management, bandwidth throttling, and web server logging. Use the [HttpCfg.exe](#) tool to add SSL certificates.

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

.NET is an open source project.
Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

Sockets in .NET

Article • 08/27/2022

The [System.Net.Sockets](#) namespace contains a managed, cross-platform socket networking implementation. All other network-access classes in the [System.Net](#) namespace are built on top of this implementation of sockets.

The [Socket](#) class is a managed-code version of the socket services provided relying on native interoperability with Linux, macOS, or Windows. In most cases, the `Socket` class methods simply marshal data into their native counterparts and handle any necessary security checks.

The `Socket` class supports two basic modes, synchronous and asynchronous. In synchronous mode, calls to functions that perform network operations (such as [SendAsync](#) and [ReceiveAsync](#)) wait until the operation completes before returning control to the calling program. In asynchronous mode, these calls return immediately.

See also

- [Use Sockets to send and receive data](#)
- [Networking in .NET](#)
- [System.Net.Sockets](#)
- [Socket](#)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

Use Sockets to send and receive data over TCP

Article • 12/01/2022

Before you can use a socket to communicate with remote devices, the socket must be initialized with protocol and network address information. The constructor for the [Socket](#) class has parameters that specify the address family, socket type, and protocol type that the socket uses to make connections. When connecting a client socket to a server socket, the client will use an [IPEndPoint](#) object to specify the network address of the server.

Create an IP endpoint

When working with [System.Net.Sockets](#), you represent a network endpoint as an [IPEndPoint](#) object. The [IPEndPoint](#) is constructed with an [IPAddress](#) and its corresponding port number. Before you can initiate a conversation through a [Socket](#), you create a data pipe between your app and the remote destination.

TCP/IP uses a network address and a service port number to uniquely identify a service. The network address identifies a specific network destination; the port number identifies the specific service on that device to connect to. The combination of network address and service port is called an endpoint, which is represented in the .NET by the [EndPoint](#) class. A descendant of [EndPoint](#) is defined for each supported address family; for the IP address family, the class is [IPEndPoint](#).

The [Dns](#) class provides domain-name services to apps that use TCP/IP internet services. The [GetHostEntryAsync](#) method queries a DNS server to map a user-friendly domain name (such as "host.contoso.com") to a numeric Internet address (such as `192.168.1.1`). [GetHostEntryAsync](#) returns a [Task<IPHostEntry>](#) that when awaited contains a list of addresses and aliases for the requested name. In most cases, you can use the first address returned in the [AddressList](#) array. The following code gets an [IPAddress](#) containing the IP address for the server `host.contoso.com`.

C#

```
IPHostEntry ipHostInfo = await Dns.GetHostEntryAsync("host.contoso.com");
IPAddress ipAddress = ipHostInfo.AddressList[0];
```

Tip

For manual testing and debugging purposes, you can typically use the `GetHostEntryAsync` method to get given the `Dns.GetHostName()` value to resolve the localhost name to an IP address.

The Internet Assigned Numbers Authority (IANA) defines port numbers for common services. For more information, see [IANA: Service Name and Transport Protocol Port Number Registry](#). Other services can have registered port numbers in the range 1,024 to 65,535. The following code combines the IP address for `host.contoso.com` with a port number to create a remote endpoint for a connection.

C#

```
IPEndPoint ipEndPoint = new(ipAddress, 11_000);
```

After determining the address of the remote device and choosing a port to use for the connection, the app can establish a connection with the remote device.

Create a `Socket` client

With the `endPoint` object created, create a client socket to connect to the server. Once the socket is connected, it can send and receive data from the server socket connection.

C#

```
using Socket client = new(
    ipEndPoint.AddressFamily,
    SocketType.Stream,
    ProtocolType.Tcp);

await client.ConnectAsync(ipEndPoint);
while (true)
{
    // Send message.
    var message = "Hi friends 🤙!<|EOM|>";
    var messageBytes = Encoding.UTF8.GetBytes(message);
    _ = await client.SendAsync(messageBytes, SocketFlags.None);
    Console.WriteLine($"Socket client sent message: \"{message}\"");

    // Receive ack.
    var buffer = new byte[1_024];
    var received = await client.ReceiveAsync(buffer, SocketFlags.None);
    var response = Encoding.UTF8.GetString(buffer, 0, received);
    if (response == "<|ACK|>")
    {
        Console.WriteLine(
            $"Socket client received acknowledgment: \"{response}\"");
```

```

        break;
    }
    // Sample output:
    //     Socket client sent message: "Hi friends 🙌!<|EOM|>"
    //     Socket client received acknowledgment: "<|ACK|>"
}

client.Shutdown(SocketShutdown.Both);

```

The preceding C# code:

- Instantiates a new `Socket` object with a given `endPoint` instances address family, the `SocketType.Stream`, and `ProtocolType.Tcp`.
- Calls the `Socket.ConnectAsync` method with the `endPoint` instance as an argument.
- In a `while` loop:
 - Encodes and sends a message to the server using `Socket.SendAsync`.
 - Writes the sent message to the console.
 - Initializes a buffer to receive data from the server using `Socket.ReceiveAsync`.
 - When the `response` is an acknowledgment, it is written to the console and the loop is exited.
- Finally, the `client` socket calls `Socket.Shutdown` given `SocketShutdown.Both`, which shuts down both send and receive operations.

Create a `Socket` server

To create the server socket, the `endPoint` object can listen for incoming connections on any IP address but the port number must be specified. Once the socket is created, the server can accept incoming connections and communicate with clients.

C#

```

using Socket listener = new(
    ipEndPoint.AddressFamily,
    SocketType.Stream,
    ProtocolType.Tcp);

listener.Bind(ipEndPoint);
listener.Listen(100);

var handler = await listener.AcceptAsync();
while (true)
{
    // Receive message.
    var buffer = new byte[1_024];

```

```

    var received = await handler.ReceiveAsync(buffer, SocketFlags.None);
    var response = Encoding.UTF8.GetString(buffer, 0, received);

    var eom = "<|EOM|>";
    if (response.IndexOf(eom) > -1 /* is end of message */)
    {
        Console.WriteLine(
            $"Socket server received message: \'{response.Replace(eom,
            "")}\'{\"}");
    }

    var ackMessage = "<|ACK|>";
    var echoBytes = Encoding.UTF8.GetBytes(ackMessage);
    await handler.SendAsync(echoBytes, 0);
    Console.WriteLine(
        $"Socket server sent acknowledgment: \'{ackMessage}\'{\"}");

    break;
}
// Sample output:
//     Socket server received message: "Hi friends 🙌!"
//     Socket server sent acknowledgment: "<|ACK|>"
}

```

The preceding C# code:

- Instantiates a new `Socket` object with a given `endPoint` instances address family, the `SocketType.Stream`, and `ProtocolType.Tcp`.
- The `listener` calls the `Socket.Bind` method with the `endPoint` instance as an argument to associate the socket with the network address.
- The `Socket.Listen()` method is called to listen for incoming connections.
- The `listener` calls the `Socket.AcceptAsync` method to accept an incoming connection on the `handler` socket.
- In a `while` loop:
 - Calls `Socket.ReceiveAsync` to receive data from the client.
 - When the data is received, it's decoded and written to the console.
 - If the `response` message ends with `<|EOM|>`, an acknowledgment is sent to the client using the `Socket.SendAsync`.

Run the sample client and server

Start the server application first, and then start the client application.

.NET CLI

```
dotnet run --project socket-server
Socket server starting...
Found: 172.23.64.1 available on port 9000.
Socket server received message: "Hi friends 🤙 !"
Socket server sent acknowledgment: "<|ACK|>"
Press ENTER to continue...
```

The client application will send a message to the server, and the server will respond with an acknowledgment.

.NET CLI

```
dotnet run --project socket-client
Socket client starting...
Found: 172.23.64.1 available on port 9000.
Socket client sent message: "Hi friends 🤙 !<|EOM|>"
Socket client received acknowledgment: "<|ACK|>"
Press ENTER to continue...
```

See also

- [Sockets in .NET](#)
- [Networking in .NET](#)
- [System.Net.Sockets](#)
- [Socket](#)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

TCP overview

Article • 12/06/2022

ⓘ Important

The [Socket](#) class is highly recommended for advanced users, instead of [TcpClient](#) and [TcpListener](#).

To work with Transmission Control Protocol (TCP), you have two options: either use [Socket](#) for maximum control and performance, or use the [TcpClient](#) and [TcpListener](#) helper classes. [TcpClient](#) and [TcpListener](#) are built on top of the [System.Net.Sockets.Socket](#) class and take care of the details of transferring data for ease of use.

The protocol classes use the underlying [Socket](#) class to provide simple access to network services without the overhead of maintaining state information or knowing the details of setting up protocol-specific sockets. To use asynchronous [Socket](#) methods, you can use the asynchronous methods supplied by the [NetworkStream](#) class. To access features of the [Socket](#) class not exposed by the protocol classes, you must use the [Socket](#) class.

[TcpClient](#) and [TcpListener](#) represent the network using the [NetworkStream](#) class. You use the [GetStream](#) method to return the network stream, and then call the stream's [NetworkStream.ReadAsync](#) and [NetworkStream.WriteAsync](#) methods. The [NetworkStream](#) does not own the protocol classes' underlying socket, so closing it does not affect the socket.

Use [TcpClient](#) and [TcpListener](#)

The [TcpClient](#) class requests data from an internet resource using TCP. The methods and properties of [TcpClient](#) abstract the details for creating a [Socket](#) for requesting and receiving data using TCP. Because the connection to the remote device is represented as a stream, data can be read and written with .NET Framework stream-handling techniques.

The TCP protocol establishes a connection with a remote endpoint and then uses that connection to send and receive data packets. TCP is responsible for ensuring that data packets are sent to the endpoint and assembled in the correct order when they arrive.

Create an IP endpoint

When working with [System.Net.Sockets](#), you represent a network endpoint as an [IPEndPoint](#) object. The [IPEndPoint](#) is constructed with an [IPAddress](#) and its corresponding port number. Before you can initiate a conversation through a [Socket](#), you create a data pipe between your app and the remote destination.

TCP/IP uses a network address and a service port number to uniquely identify a service. The network address identifies a specific network destination; the port number identifies the specific service on that device to connect to. The combination of network address and service port is called an endpoint, which is represented in the .NET by the [EndPoint](#) class. A descendant of [EndPoint](#) is defined for each supported address family; for the IP address family, the class is [IPEndPoint](#).

The [Dns](#) class provides domain-name services to apps that use TCP/IP internet services. The [GetHostEntryAsync](#) method queries a DNS server to map a user-friendly domain name (such as "host.contoso.com") to a numeric Internet address (such as [192.168.1.1](#)). [GetHostEntryAsync](#) returns a [Task<IPHostEntry>](#) that when awaited contains a list of addresses and aliases for the requested name. In most cases, you can use the first address returned in the [AddressList](#) array. The following code gets an [IPAddress](#) containing the IP address for the server [host.contoso.com](#).

C#

```
IPHostEntry ipHostInfo = await Dns.GetHostEntryAsync("host.contoso.com");
IPAddress ipAddress = ipHostInfo.AddressList[0];
```

Tip

For manual testing and debugging purposes, you can typically use the [GetHostEntryAsync](#) method to get given the [Dns.GetHostName\(\)](#) value to resolve the localhost name to an IP address.

The Internet Assigned Numbers Authority (IANA) defines port numbers for common services. For more information, see [IANA: Service Name and Transport Protocol Port Number Registry](#). Other services can have registered port numbers in the range 1,024 to 65,535. The following code combines the IP address for [host.contoso.com](#) with a port number to create a remote endpoint for a connection.

C#

```
IPEndPoint ipEndPoint = new(ipAddress, 11_000);
```

After determining the address of the remote device and choosing a port to use for the connection, the app can establish a connection with the remote device.

Create a `TcpClient`

The `TcpClient` class provides TCP services at a higher level of abstraction than the `Socket` class. `TcpClient` is used to create a client connection to a remote host. Knowing how to get an `IPPEndPoint`, let's assume you have an `IPAddress` to pair with your desired port number. The following example demonstrates setting up a `TcpClient` to connect to a time server on TCP port 13:

C#

```
var ipEndPoint = new IPPEndPoint(ipAddress, 13);

using TcpClient client = new();
await client.ConnectAsync(ipEndPoint);
await using NetworkStream stream = client.GetStream();

var buffer = new byte[1_024];
int received = await stream.ReadAsync(buffer);

var message = Encoding.UTF8.GetString(buffer, 0, received);
Console.WriteLine($"Message received: \"{message}\"");
// Sample output:
//     Message received: "📅 8/22/2022 9:07:17 AM ⏳"
```

The preceding C# code:

- Creates an `IPPEndPoint` from a known `IPAddress` and port.
- Instantiate a new `TcpClient` object.
- Connects the `client` to the remote TCP time server on port 13 using `TcpClient.ConnectAsync`.
- Uses a `NetworkStream` to read data from the remote host.
- Declares a read buffer of `1_024` bytes.
- Reads data from the `stream` into the read buffer.
- Writes the results as a string to the console.

Since the client knows that the message is small, the entire message can be read into the read buffer in one operation. With larger messages, or messages with an

indeterminate length, the client should use the buffer more appropriately and read in a `while` loop.

ⓘ Important

When sending and receiving messages, the [Encoding](#) should be known ahead of time to both server and client. For example, if the server communicates using [ASCIIEncoding](#) but the client attempts to use [UTF8Encoding](#), the messages will be malformed.

Create a `TcpListener`

The [TcpListener](#) type is used to monitor a TCP port for incoming requests and then create either a [Socket](#) or a [TcpClient](#) that manages the connection to the client. The [Start](#) method enables listening, and the [Stop](#) method disables listening on the port. The [AcceptTcpClientAsync](#) method accepts incoming connection requests and creates a [TcpClient](#) to handle the request, and the [AcceptSocketAsync](#) method accepts incoming connection requests and creates a [Socket](#) to handle the request.

The following example demonstrates creating a network time server using a `TcpListener` to monitor TCP port 13. When an incoming connection request is accepted, the time server responds with the current date and time from the host server.

C#

```
var ipEndPoint = new IPEndPoint(IPAddress.Any, 13);
TcpListener listener = new(ipEndPoint);

try
{
    listener.Start();

    using TcpClient handler = await listener.AcceptTcpClientAsync();
    await using NetworkStream stream = handler.GetStream();

    var message = $"📅 {DateTime.Now} ⏰";
    var dateTimeBytes = Encoding.UTF8.GetBytes(message);
    await stream.WriteAsync(dateTimeBytes);

    Console.WriteLine($"Sent message: \'{message}\'");
    // Sample output:
    //      Sent message: "📅 8/22/2022 9:07:17 AM ⏰"
}
finally
{}
```

```
    listener.Stop();
}
```

The preceding C# code:

- Creates an `IPPEndPoint` with `IPAddress.Any` and port.
- Instantiate a new `TcpListener` object.
- Calls the `Start` method to start listening on the port.
- Uses a `TcpClient` from the `AcceptTcpClientAsync` method to accept incoming connection requests.
- Encodes the current date and time as a string message.
- Uses a `NetworkStream` to write data to the connected client.
- Writes the sent message to the console.
- Finally, calls the `Stop` method to stop listening on the port.

Finite TCP control with the `Socket` class

Both `TcpClient` and `TcpListener` internally rely on the `Socket` class, meaning anything you can do with these classes can be achieved using sockets directly. This section demonstrates several `TcpClient` and `TcpListener` use cases, along with their `Socket` counterpart that is functionally equivalent.

Create a client socket

`TcpClient`'s default constructor tries to create a *dual-stack socket* via the `Socket(SocketType, ProtocolType)` constructor. This constructor creates a dual-stack socket if [IPv6 is supported](#), otherwise, it falls back to IPv4.

Consider the following TCP client code:

```
C#
```

```
using var client = new TcpClient();
```

The preceding TCP client code is functionally equivalent to the following socket code:

```
C#
```

```
using var socket = new Socket(SocketType.Stream, ProtocolType.Tcp);
```

The [TcpClient\(AddressFamily\)](#) constructor

This constructor accepts only three `AddressFamily` values, otherwise it will throw a `ArgumentException`. Valid values are:

- `AddressFamily.InterNetwork`: for IPv4 socket.
- `AddressFamily.InterNetworkV6`: for IPv6 socket.
- `AddressFamily.Unknown`: this will attempt to create a dual-stack socket, similarly to the default constructor.

Consider the following TCP client code:

C#

```
using var client = new TcpClient(AddressFamily.InterNetwork);
```

The preceding TCP client code is functionally equivalent to the following socket code:

C#

```
using var socket = new Socket(AddressFamily.InterNetwork, SocketType.Stream,  
ProtocolType.Tcp);
```

The [TcpClient\(IPEndPoint\)](#) constructor

Upon creating the socket, this constructor will also **bind** to the provided `local IPEndPoint`. The `IPEndPoint.AddressFamily` property is used to determine the address family of the socket.

Consider the following TCP client code:

C#

```
var endPoint = new IPEndPoint(IPAddress.Parse("127.0.0.1"), 5001);  
using var client = new TcpClient(endPoint);
```

The preceding TCP client code is functionally equivalent to the following socket code:

C#

```
// Example IPEndPoint object  
var endPoint = new IPEndPoint(IPAddress.Parse("127.0.0.1"), 5001);  
using var socket = new Socket(endPoint.AddressFamily, SocketType.Stream,
```

```
ProtocolType.Tcp);  
socket.Bind(endPoint);
```

The `TcpClient(String, Int32)` constructor

This constructor will attempt to create a dual-stack similar to the default constructor and connect it to the **remote** DNS endpoint defined by the `hostname` and `port` pair.

Consider the following TCP client code:

C#

```
using var client = new TcpClient("www.example.com", 80);
```

The preceding TCP client code is functionally equivalent to the following socket code:

C#

```
using var socket = new Socket(SocketType.Stream, ProtocolType.Tcp);  
socket.Connect("www.example.com", 80);
```

Connect to server

All `Connect`, `ConnectAsync`, `BeginConnect` and `EndConnect` overloads in `TcpClient` are functionally equivalent to the corresponding `Socket` methods.

Consider the following TCP client code:

C#

```
using var client = new TcpClient();  
client.Connect("www.example.com", 80);
```

The above `TcpClient` code is equivalent to the following socket code:

C#

```
using var socket = new Socket(SocketType.Stream, ProtocolType.Tcp);  
socket.Connect("www.example.com", 80);
```

Create a server socket

Much like `TcpClient` instances having functional equivalency with their raw `Socket` counterparts, this section maps `TcpListener` constructors to their corresponding socket code. The first constructor to consider is the `TcpListener(IPAddress localaddr, int port)`.

```
C#
```

```
var listener = new TcpListener(IPAddress.Loopback, 5000);
```

The preceding TCP listener code is functionally equivalent to the following socket code:

```
C#
```

```
var ep = new IPEndPoint(IPAddress.Loopback, 5000);
using var socket = new Socket(ep.AddressFamily, SocketType.Stream,
ProtocolType.Tcp);
```

Start listening on the server

The `Start()` method is a wrapper combining `Socket`'s `Bind` and `Listen()` functionality.

Consider the following TCP listener code:

```
C#
```

```
var listener = new TcpListener(IPAddress.Loopback, 5000);
listener.Start(10);
```

The preceding TCP listener code is functionally equivalent to the following socket code:

```
C#
```

```
var endPoint = new IPEndPoint(IPAddress.Loopback, 5000);
using var socket = new Socket(endPoint.AddressFamily, SocketType.Stream,
ProtocolType.Tcp);
socket.Bind(endPoint);
try
{
    socket.Listen(10);
}
catch (SocketException)
{
    socket.Dispose();
}
```

Accept a server connection

Under the hood, incoming TCP connections are always creating a new socket when accepted. `TcpListener` can accept a `Socket` instance directly (via `AcceptSocket()` or `AcceptSocketAsync()`) or it can accept a `TcpClient` (via `AcceptTcpClient()` and `AcceptTcpClientAsync()`).

Consider the following `TcpListener` code:

C#

```
var listener = new TcpListener(IPAddress.Loopback, 5000);
using var acceptedSocket = await listener.AcceptSocketAsync();

// Synchronous alternative.
// var acceptedSocket = listener.AcceptSocket();
```

The preceding TCP listener code is functionally equivalent to the following socket code:

C#

```
var endPoint = new IPEndPoint(IPAddress.Loopback, 5000);
using var socket = new Socket(endPoint.AddressFamily, SocketType.Stream,
ProtocolType.Tcp);
using var acceptedSocket = await socket.AcceptAsync();

// Synchronous alternative
// var acceptedSocket = socket.Accept();
```

Create a `NetworkStream` to send and receive data

With `TcpClient` you need to instantiate a `NetworkStream` with the `GetStream()` method to be able to send and receive data . With `Socket`, you have to do the `NetworkStream` creation manually.

Consider the following `TcpClient` code:

C#

```
using var client = new TcpClient();
using NetworkStream stream = client.GetStream();
```

Which is equivalent to the following socket code:

C#

```
using var socket = new Socket(SocketType.Stream, ProtocolType.Tcp);

// Be aware that transferring the ownership means that closing/disposing the
// stream will also close the underlying socket.
using var stream = new NetworkStream(socket, ownsSocket: true);
```

💡 Tip

If your code doesn't need to work with a `Stream` instance, you can rely on `Socket`'s Send/Receive methods (`Send`, `SendAsync`, `Receive` and `ReceiveAsync`) directly instead of creating a `NetworkStream`.

See also

- [Use Sockets to send and receive data over TCP](#)
- [Networking in .NET](#)
- [Socket](#)
- [TcpClient](#)
- [TcpListener](#)
- [NetworkStream](#)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

System.Net.Sockets.Socket class

Article • 01/08/2024

This article provides supplementary remarks to the reference documentation for this API.

The [Socket](#) class provides a rich set of methods and properties for network communications. The [Socket](#) class allows you to perform both synchronous and asynchronous data transfer using any of the communication protocols listed in the [ProtocolType](#) enumeration.

The [Socket](#) class follows the .NET naming pattern for asynchronous methods. For example, the synchronous [Receive](#) method corresponds to the asynchronous [ReceiveAsync](#) variants.

Use the following methods for synchronous operation mode:

- If you are using a connection-oriented protocol such as TCP, your server can listen for connections using the [Listen](#) method. The [Accept](#) method processes any incoming connection requests and returns a [Socket](#) that you can use to communicate data with the remote host. Use this returned [Socket](#) to call the [Send](#) or [Receive](#) method. Call the [Bind](#) method prior to calling the [Listen](#) method if you want to specify the local IP address and port number. Use a port number of zero if you want the underlying service provider to assign a free port for you. If you want to connect to a listening host, call the [Connect](#) method. To communicate data, call the [Send](#) or [Receive](#) method.
- If you are using a connectionless protocol such as UDP, you do not need to listen for connections at all. Call the [ReceiveFrom](#) method to accept any incoming datagrams. Use the [SendTo](#) method to send datagrams to a remote host.

To process communications asynchronously, use the following methods:

- If you are using a connection-oriented protocol such as TCP, use [ConnectAsync](#) to connect with a listening host. Use [SendAsync](#) or [ReceiveAsync](#) to communicate data asynchronously. Incoming connection requests can be processed using [AcceptAsync](#).
- If you are using a connectionless protocol such as UDP, you can use [SendToAsync](#) to send datagrams, and [ReceiveFromAsync](#) to receive datagrams.

If you perform multiple asynchronous operations on a socket, they do not necessarily complete in the order in which they are started.

When you are finished sending and receiving data, use the [Shutdown](#) method to disable the [Socket](#). After calling [Shutdown](#), call the [Close](#) method to release all resources associated with the [Socket](#).

The [Socket](#) class allows you to configure your [Socket](#) using the [SetSocketOption](#) method. Retrieve these settings using the [GetSocketOption](#) method.

 [Collaborate with us on GitHub](#)

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

 .NET

[.NET feedback](#)

.NET is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

WebSockets support in .NET

Article • 03/25/2023

The WebSocket protocol enables two-way communication between a client and a remote host. The [System.Net.WebSockets.ClientWebSocket](#) exposes the ability to establish WebSocket connection via an opening handshake, it is created and sent by the `ConnectAsync` method.

Differences in HTTP/1.1 and HTTP/2 WebSockets

WebSockets over HTTP/1.1 uses a single TCP connection, therefore it is managed by connection-wide headers, for more information, see [RFC 6455](#). Consider the following example of how to establish WebSocket over HTTP/1.1:

```
c#  
  
Uri uri = new("ws://corefx-net-  
http11.azurewebsites.net/WebSocket/EchoWebSocket.ashx");  
  
using ClientWebSocket ws = new();  
await ws.ConnectAsync(uri, default);  
  
var bytes = new byte[1024];  
var result = await ws.ReceiveAsync(bytes, default);  
string res = Encoding.UTF8.GetString(bytes, 0, result.Count);  
  
await ws.CloseAsync(WebSocketCloseStatus.NormalClosure, "Client closed",  
default);
```

A different approach must be taken with HTTP/2 due to its multiplexing nature. WebSockets are established per stream, for more information, see [RFC 8441](#). With HTTP/2 it is possible to use one connection for multiple web socket streams together with ordinary HTTP streams and extend HTTP/2's more efficient use of the network to WebSockets. There is a special overload of [ConnectAsync\(Uri, HttpMessageInvoker, CancellationToken\)](#) which accepts an [HttpMessageInvoker](#) to allow reusing existing pooled connections:

```
c#  
  
using SocketsHttpHandler handler = new();  
using ClientWebSocket ws = new();
```

```
await ws.ConnectAsync(uri, new HttpMessageInvoker(handler),  
cancellationToken);
```

Set up HTTP version and policy

By default, `ClientWebSocket` uses HTTP/1.1 to send an opening handshake and allows downgrade. In .NET 7 web sockets over HTTP/2 are available. It can be changed before calling `ConnectAsync`:

```
c#  
  
using SocketsHttpHandler handler = new();  
using ClientWebSocket ws = new();  
  
ws.Options.HttpVersion = HttpVersion.Version20;  
ws.Options.HttpVersionPolicy = HttpVersionPolicy.RequestVersionOrHigher;  
  
await ws.ConnectAsync(uri, new HttpMessageInvoker(handler),  
cancellationToken);
```

Incompatible options

`ClientWebSocket` has properties `System.Net.WebSockets.ClientWebSocketOptions` that the user can set up before the connection is established. However, when `HttpMessageInvoker` is provided, it also has these properties. To avoid ambiguity, in that case, properties should be set on `HttpMessageInvoker`, and `ClientWebSocketOptions` should have default values. Otherwise, if `ClientWebSocketOptions` are changed, overload of `ConnectAsync` will throw an [ArgumentException](#).

```
c#  
  
using HttpClientHandler handler = new()  
{  
    CookieContainer = cookies;  
    UseCookies = cookies != null;  
    ServerCertificateCustomValidationCallback =  
    remoteCertificateValidationCallback;  
    Credentials = useDefaultCredentials  
        ? CredentialCache.DefaultCredentials  
        : credentials;  
};  
if (proxy is null)  
{  
    handler.UseProxy = false;  
}
```

```
else
{
    handler.Proxy = proxy;
}
if (clientCertificates?.Count > 0)
{
    handler.ClientCertificates.AddRange(clientCertificates);
}
HttpMessageInvoker invoker = new(handler);
using ClientWebSocket cws = new();
await cws.ConnectAsync(uri, invoker, cancellationToken);
```

Compression

The WebSocket protocol supports per-message deflate as defined in [RFC 7692](#). It is controlled by

[System.Net.WebSockets.ClientWebSocketOptions.DangerousDeflateOptions](#). When present, the options are sent to the server during the handshake phase. If the server supports per-message-deflate and the options are accepted, the `ClientWebSocket` instance will be created with compression enabled by default for all messages.

```
c#

using ClientWebSocket ws = new()
{
    Options =
    {
        DangerousDeflateOptions = new WebSocketDeflateOptions()
        {
            ClientMaxWindowBits = 10,
            ServerMaxWindowBits = 10
        }
    }
};
```

ⓘ Important

Before using compression, please be aware that enabling it makes the application subject to CRIME/BREACH type of attacks, for more information, see [CRIME](#) and [BREACH](#). It is strongly advised to turn off compression when sending data containing secrets by specifying the `DisableCompression` flag for such messages.

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

TLS/SSL best practices

Article • 03/18/2023

TLS (Transport Layer Security) is a cryptographic protocol designed to secure communication between two computers over the internet. The TLS protocol is exposed in .NET via the [SslStream](#) class.

This article presents best practices for setting up secure communication between client and server and assumes use of .NET. For best practices with .NET Framework, see [Transport Layer Security \(TLS\) best practices with the .NET Framework](#).

Select TLS version

While it is possible to specify the version of the TLS protocol to be used via the [EnabledSslProtocols](#) property, it is recommended to defer to the operating system settings by using [None](#) value (this is the default).

Deferring the decision to the OS automatically uses the most recent version of TLS available and lets the application pick up changes after OS upgrades. The operating system may also prevent use of TLS versions which are no longer considered secure.

Select cipher suites

`SslStream` allows users to specify which cipher suites can be negotiated by the TLS handshake via the [CipherSuitesPolicy](#) class. As with TLS versions, it's recommended to let the OS decide which are the best cipher suites to negotiate with, and, therefore, it's recommended to avoid using [CipherSuitesPolicy](#).

Note

`CipherSuitesPolicy` is not supported on Windows and attempts to instantiate it will cause `NotSupportedException` to be thrown.

Specify a server certificate

When authenticating as a server, `SslStream` requires an [X509Certificate2](#) instance. It is recommended to always use an [X509Certificate2](#) instance which also contains the private key.

There are multiple ways that a server certificate can be passed to `SslStream`:

- Directly as a parameter to `SslStream.AuthenticateAsServerAsync` or via `SslServerAuthenticationOptions.ServerCertificate` property
- From a selection callback in `SslServerAuthenticationOptions.ServerCertificateSelectionCallback` property
- By passing a `SslStreamCertificateContext` in the `SslServerAuthenticationOptions.ServerCertificateContext` property

The recommended approach is to use the `SslServerAuthenticationOptions.ServerCertificateContext` property. When the certificate is obtained by one of the other two ways, a `SslStreamCertificateContext` instance is created internally by the `SslStream` implementation. Creating a `SslStreamCertificateContext` involves building an `X509Chain` which is a CPU intensive operation. It is more efficient to create a `SslStreamCertificateContext` once and reuse it for multiple `SslStream` instances.

Reusing `SslStreamCertificateContext` instances also enables additional features such as [TLS session resumption](#) on Linux servers.

Custom X509Certificate validation

There are certain scenarios in which the default certificate validation procedure isn't adequate and some custom validation logic is required. Parts of the validation logic can be customized by specifying `SslClientAuthenticationOptions.CertificateChainPolicy` or `SslServerAuthenticationOptions.CertificateChainPolicy`. Alternatively, completely custom logic can be provided via the `<System.Net.Security.SslClientAuthenticationOptions.RemoteCertificateValidationCallback>` property. For more information, see [Custom certificate trust](#).

Custom certificate trust

When encountering a certificate that wasn't issued by any of the certificate authorities trusted by the machine (including self-signed certificates), the default certificate validation procedure will fail. One possible way to resolve this is to add the necessary issuer certificates to the machine's trusted store. That, however, might affect other applications on the system and is not always possible.

The alternative solution is to specify custom trusted root certificates via an `X509ChainPolicy`. To specify a custom trust list that will be used instead of the system trust list during validation, consider the following example:

C#

```
SslClientAuthenticationOptions clientOptions = new();

clientOptions.CertificateChainPolicy = new X509ChainPolicy()
{
    TrustMode = X509ChainTrustMode.CustomRootTrust,
    CustomTrustStore =
    {
        customIssuerCert
    }
};
```

Clients configured with the preceding policy would only accept certificates trusted by `customIssuerCert`.

Ignore specific validation errors

Consider an IoT device without a persistent clock. After powering on, the clock of the device would start many years in the past and, therefore, all certificates would be considered "not yet valid". Consider the following code that shows a validation callback implementation ignoring validity period violations.

C#

```
static bool CustomCertificateValidationCallback(
    object sender,
    X509Certificate? certificate,
    X509Chain? chain,
    SslPolicyErrors sslPolicyErrors)
{
    // Anything that would have been accepted by default is OK
    if (sslPolicyErrors == SslPolicyErrors.None)
    {
        return true;
    }

    // If there is something wrong other than a chain processing error,
    // don't trust it.
    if (sslPolicyErrors != SslPolicyErrors.RemoteCertificateChainErrors)
    {
        return false;
    }

    Debug.Assert(chain is not null);

    foreach (X509ChainStatus status in chain.ChainStatus)
    {
        // If an error other than `NotTimeValid` (or `NoError`) is present,
        // don't trust it.
```

```

    if ((status.Status & ~X509ChainStatusFlags.NotTimeValid) != X509ChainStatusFlags.NoError)
    {
        return false;
    }
}

return true;
}

```

Certificate pinning

Another situation where custom certificate validation is necessary is when clients expect servers to use a specific certificate, or a certificate from a small set of known certificates. This practice is known as [certificate pinning](#). The following code snippet shows a validation callback which checks that the server presents a certificate with a specific known public key.

C#

```

static bool CustomCertificateValidationCallback(
    object sender,
    X509Certificate? certificate,
    X509Chain? chain,
    SslPolicyErrors sslPolicyErrors)
{
    // If there is something wrong other than a chain processing error,
    // don't trust it.
    if ((sslPolicyErrors & ~SslPolicyErrors.RemoteCertificateChainErrors) != 0)
    {
        return false;
    }

    Debug.Assert(certificate is not null);

    const string ExpectedPublicKey =
        "3082010A0282010100C204ECF88CEE04C2B3D850D57058CC9318EB5C" +
        "A86849B022B5F9959EB12B2C763E6CC04B604C4CEAB2B4C00F80B6B0" +
        "F972C98602F95C415D132B7F71C44BBC9942E5037A6671C618CF641" +
        "42C546D31687279F74EB0A9D11522621736C844C7955E4D16BE8063D" +
        "481552ADB328DBAAFF6EFF60954A776B39F124D131B6DD4DC0C4FC53" +
        "B96D42ADB57CFEAFF515D23348E72271C7C2147A6C28EA374ADFEA6C" +
        "B572B47E5AA216DC69B15744DB0A12ABDEC30F47745C4122E19AF91B" +
        "93E6AD2206292EB1BA491C0C279EA3FB8BF7407200AC9208D98C5784" +
        "538105CBE6FE6B5498402785C710BB7370EF6918410745557CF9643F" +
        "3D2CC3A97CEB931A4C86D1CA850203010001";

    return certificate.GetPublicKeyString().Equals(ExpectedPublicKey);
}

```

Considerations for client certificate validation

Server applications need to be careful when requiring and validating client certificates. Certificates may contain the [AIA \(Authority Information Access\)](#) ↗ extension which specifies where the issuer certificate can be downloaded. The server may therefore attempt to download the issuer certificate from external server when building the [X509Chain](#) for the client certificate. Similarly, servers may need to contact external servers to ensure that the client certificate has not been revoked.

The need to contact external servers when building and validating the [X509Chain](#) may expose the application to denial of service attacks if the external servers are slow to respond. Therefore, server applications should configure the [X509Chain](#) building behavior using the [CertificateChainPolicy](#).

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

Troubleshoot `SslStream` authentication issues

Article • 03/18/2023

This article presents the most frequent authentication issues when using `SslStream` cryptography- and security-related functionalities in .NET are implemented by interop with either the OS API (such as Schannel on Windows) or low-level system libraries (like OpenSSL on Linux). The behavior of .NET application, including exception messages and error codes may therefore change depending on which platform it is run.

Some issues may be easier to investigate and troubleshoot by observing the actual messages exchanged over the wire using tools such as [Wireshark](#) or [tcpdump](#). These tools can be used to inspect the `ClientHello`, `ServerHello`, and other messages for advertised supported TLS versions allowed and negotiated cipher suites and the certificates exchanged for authentication.

Intermediate certificates are not sent

During the TLS handshake, the server (and the client too, if client authentication is requested) sends its certificate to prove its identity to the client. In order to validate the authenticity of the certificate, a chain of certificates needs to be built and verified. The root of the chain must be one of the trusted root certificate authority (CA), the certificate of which is stored in the machine certificate store.

If the peer certificate hasn't been issued by one of the trusted CAs an intermediate CA certificate is necessary to build the certificate chain. However, if the intermediate certificate isn't available, it isn't possible to validate the certificate and the TLS handshake may fail.

This issue is most frequently encountered on Windows. Even though the application provided intermediate certificates via the authentication options, they will not be sent to the peer unless they are stored in the Windows certificate store. This limitation is due to the fact that the actual TLS handshake occurs outside of the application process.

For server applications, it is possible to pass an `SslStreamCertificateContext` as `SslServerAuthenticationOptions.ServerCertificateContext`. During construction of the `SslStreamCertificateContext` instance, you can pass additional intermediate certificates and these will be temporarily added into the certificate store.

Unfortunately, for client application the only solution is to add the certificates to the certificate store manually.

Handshake failed with ephemeral keys

On Windows, you may encounter the `(0x8009030E): No credentials are available in the security package` error message when attempting to use certificates with ephemeral keys. This behavior is due to a bug in the underlying OS API (Schannel). More relevant info and workarounds can be found on the associated [GitHub issue](#).

Client and server do not possess a common algorithm

When inspecting the `ClientHello` and `ServerHello` messages, you may find out that there is no cipher suite offered by both client and server or even that some ciphers are not offered even if explicitly configured via `CipherSuitesPolicy` (available on Linux only). The underlying TLS library may disable TLS versions and cipher suites which are considered insecure.

On many Linux distributions, the relevant configuration file is located at `/etc/ssl/openssl.cnf`.

On Windows, the `Enable-TlsCipherSuite` and `Disable-TlsCipherSuite` PowerShell cmdlets can be used to configure cipher suites. Individual TLS versions can be enabled/disable by configuring the

`HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\SecurityProviders\SCHANNEL\Protocols\TLS <version>\{Client|Server}\Enabled` registry key.

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



.NET feedback

The .NET documentation is open source. Provide feedback [here](#).

 [Open a documentation issue](#)

 [Provide product feedback](#)

Migrate SslStream code from .NET Framework to .NET

Article • 03/18/2023

.NET Core brought many improvements as well as breaking changes to how [SslStream](#) works. The most important change related to network security is that the [ServicePointManager](#) class has been mostly obsoleted and affects only the legacy [WebRequest](#) interface.

For each [SslStream](#) instance, you must configure the allowed TLS protocols and certificate validation callbacks separately via [SslServerAuthenticationOptions](#) or [SslClientAuthenticationOptions](#). To configure network security options used in HTTPS in [HttpClient](#), you need to configure the security options in the underlying handler. The default handler used by [HttpClient](#) is [SocketsHttpHandler](#), which has an [SslOptions](#) property that accepts [SslClientAuthenticationOptions](#).

Consider the following example that demonstrates how to create an [HttpClient](#) with a custom certificate validation callback:

C#

```
bool CustomCertificateValidator(
    object sender,
    X509Certificate? certificate,
    X509Chain? chain,
    SslPolicyErrors sslPolicyErrors)
{
    // TODO: Always returns false.
    // Need to implement certificate evaluation logic.
    return false;
}

HttpClient httpClient = new(
    new SocketsHttpHandler
    {
        SslOptions =
        {
            RemoteCertificateValidationCallback = CustomCertificateValidator
        }
    });

```

The following table shows how to migrate individual [ServicePointManager](#) properties related to TLS.

| Source API | Target API |
|-------------------------------------|---|
| CheckCertificateRevocationList | Set appropriate X509RevocationMode on SslClientAuthenticationOptions.CertificateRevocationCheckMode . |
| EncryptionPolicy | Use SslClientAuthenticationOptions.EncryptionPolicy . |
| SecurityProtocol | Use SslClientAuthenticationOptions.EnabledSslProtocols . |
| ServerCertificateValidationCallback | Use SslClientAuthenticationOptions.RemoteCertificateValidationCallback . |

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

QUIC protocol

Article • 12/30/2023

QUIC is a network transport layer protocol standardized in [RFC 9000](#). It uses UDP as an underlying protocol and it's inherently secure as it mandates TLS 1.3 usage. For more information, see [RFC 9001](#). Another interesting difference from well-known transport protocols such as TCP and UDP is that it has stream multiplexing built-in on the transport layer. This allows having multiple, concurrent, independent data streams that don't affect each other.

QUIC itself doesn't define any semantics for the exchanged data as it's a transport protocol. It's rather used in application layer protocols, for example in [HTTP/3](#) or in [SMB over QUIC](#). It can also be used for any custom-defined protocol.

The protocol offers many advantages over TCP with TLS, here are a few:

- Faster connection establishment as it doesn't require as many round trips as TCP with TLS on top.
- Avoidance of head-of-line blocking problem where one lost packet doesn't block data of all the other streams.

On the other hand, there are potential disadvantages to consider when using QUIC. As a newer protocol, its adoption is still growing and limited. Apart from that, QUIC traffic may be even blocked by some networking components.

QUIC in .NET

The QUIC implementation was introduced in .NET 5 as the `System.Net.Quic` library. However, up until .NET 7.0 the library was strictly internal and served only as an implementation of HTTP/3. With .NET 7, the library was made public thus exposing its APIs.

ⓘ Note

In .NET 7.0, the APIs are published as [preview features](#).

From the implementation perspective, `System.Net.Quic` depends on [MsQuic](#), the native implementation of QUIC protocol. As a result, `System.Net.Quic` platform support and dependencies are inherited from MsQuic and documented in the [Platform dependencies](#) section. In short, the MsQuic library is shipped as part of .NET for Windows. But for Linux, you must manually install `libmsquic` via an appropriate package manager. For the other platforms, it's still possible to build MsQuic manually, whether against SChannel or OpenSSL, and use it with `System.Net.Quic`. However, these scenarios are not part of our testing matrix and unforeseen problems might occur.

Platform dependencies

The following sections describe the platform dependencies for QUIC in .NET.

Windows

- Windows 11, Windows Server 2022, or later. (Earlier Windows versions are missing the cryptographic APIs required to support QUIC.)

On Windows, msquic.dll is distributed as part of the .NET runtime, and no other steps are required to install it.

Linux

Note

.NET 7+ is only compatible with 2.2+ versions of libmsquic.

The `libmsquic` package is required on Linux. This package is published in Microsoft's official Linux package repository, <https://packages.microsoft.com>. You must add this repository to your package manager before installing the package. For more information, see [Linux Software Repository for Microsoft Products](#).

Caution

Adding the Microsoft pacakge repository may conflict with your distribution's repository when your distribution's repository provides .NET and other Microsoft packages. To avoid or troubleshoot package mixups, review [Troubleshoot .NET errors related to missing files on Linux](#).

Examples

Here are some examples of using a package manager to install `libmsquic`:

- APT

```
Bash
```

```
sudo apt-get libmsquic
```

- APK

```
Bash
```

```
sudo apk add libmsquic
```

- DNF

```
Bash
```

```
sudo dnf install libmsquic
```

- zypper

```
Bash
```

```
sudo zypper install libmsquic
```

- YUM

```
Bash
```

```
sudo yum install libmsquic
```

Dependencies of libmsquic

All the following dependencies are stated in the `libmsquic` package manifest and are automatically installed by the package manager:

- OpenSSL 3+ or 1.1 - depends on the default OpenSSL version for the distribution version, for example, OpenSSL 3 for [Ubuntu 22](#) and OpenSSL 1.1 for [Ubuntu 20](#).
- libnuma 1

macOS

QUIC isn't currently supported on macOS but may be available in a future release.

API overview

[System.Net.Quic](#) brings three major classes that enable the usage of QUIC protocol:

- [QuicListener](#) - server side class for accepting incoming connections.
- [QuicConnection](#) - QUIC connection, corresponding to [RFC 9000 Section 5](#).
- [QuicStream](#) - QUIC stream, corresponding to [RFC 9000 Section 2](#).

But before using these classes, your code should check whether QUIC is currently supported, as `libmsquic` might be missing, or TLS 1.3 might not be supported. For that, both `QuicListener` and `QuicConnection` expose a static property `IsSupported`:

```
C#
```

```
if (QuicListener.IsSupported)
{
    // Use QuicListener
}
else
{
    // Fallback/Error
}

if (QuicConnection.IsSupported)
{
    // Use QuicConnection
}
else
{
    // Fallback/Error
}
```

These properties will report the same value, but that might change in the future. It's recommended to check [IsSupported](#) for server-scenarios and [IsSupported](#) for the client ones.

QuicListener

[QuicListener](#) represents a server side class that accepts incoming connections from the clients. The listener is constructed and started with a static method [ListenAsync\(QuicListenerOptions, CancellationToken\)](#). The method accepts an instance of [QuicListenerOptions](#) class with all the settings necessary to start the listener and accept incoming connections. After that, listener is ready to hand out connections via [AcceptConnectionAsync\(CancellationToken\)](#). Connections returned by this method are always fully connected, meaning that the TLS handshake is finished and the connection is ready to be used. Finally, to stop listening and release all resources, [DisposeAsync\(\)](#) must be called.

Consider the following [QuicListener](#) example code:

C#

```
using System.Net.Quic;

// First, check if QUIC is supported.
if (!QuicListener.IsSupported)
{
    Console.WriteLine("QUIC is not supported, check for presence of libmsquic and support of
TLS 1.3.");
    return;
}

// Share configuration for each incoming connection.
// This represents the minimal configuration necessary.
var serverConnectionOptions = new QuicServerConnectionOptions
{
    // Used to abort stream if it's not properly closed by the user.
    // See https://www.rfc-editor.org/rfc/rfc9000#section-20.2
    DefaultStreamErrorCode = 0x0A, // Protocol-dependent error code.

    // Used to close the connection if it's not done by the user.
    // See https://www.rfc-editor.org/rfc/rfc9000#section-20.2
    DefaultCloseErrorCode = 0x0B, // Protocol-dependent error code.

    // Same options as for server side SslStream.
    ServerAuthenticationOptions = new SslServerAuthenticationOptions
    {
        // List of supported application protocols, must be the same or subset of
        QuicListenerOptions.ApplicationProtocols.
        ApplicationProtocols = new List<SslApplicationProtocol>() { "protocol-name" },
        // Server certificate, it can also be provided via ServerCertificateContext or
        ServerCertificateSelectionCallback.
        ServerCertificate = serverCertificate
    }
};

// Initialize, configure the listener and start listening.
var listener = await QuicListener.ListenAsync(new QuicListenerOptions
{
    // Listening endpoint, port 0 means any port.
    ListenEndPoint = new IPPEndPoint(IPAddress.Loopback, 0),
    // List of all supported application protocols by this listener.
    ApplicationProtocols = new List<SslApplicationProtocol>() { "protocol-name" },
    // Callback to provide options for the incoming connections, it gets called once per each
    connection.
});
```

```

    ConnectionOptionsCallback = (_, _, _) => ValueTask.FromResult(serverConnectionOptions)
});

// Accept and process the connections.
while (isRunning)
{
    // Accept will propagate any exceptions that occurred during the connection
    // establishment,
    // including exceptions thrown from ConnectionOptionsCallback, caused by invalid
    QuicServerConnectionOptions or TLS handshake failures.
    var connection = await listener.AcceptConnectionAsync();

    // Process the connection...
}

// When finished, dispose the listener.
await listener.DisposeAsync();

```

For more information about how the `QuicListener` was designed, see the [API proposal ↗](#).

QuicConnection

`QuicConnection` is a class used for both server and client side QUIC connections. Server side connections are created internally by the listener and handed out via `AcceptConnectionAsync(CancellationToken)`. Client side connections must be opened and connected to the server. As with the listener, there's a static method `ConnectAsync(QuicClientConnectionOptions, CancellationToken)` that instantiates and connects the connection. It accepts an instance of `QuicClientConnectionOptions`, an analogous class to `QuicServerConnectionOptions`. After that, the work with the connection doesn't differ between client and server. It can open outgoing streams and accept incoming ones. It also provides properties with information about the connection, like `LocalEndPoint`, `RemoteEndPoint`, or `RemoteCertificate`.

When the work with the connection is done, it needs to be closed and disposed. QUIC protocol mandates using an application layer code for immediate closure, see [RFC 9000 Section 10.2 ↗](#). For that, `CloseAsync(Int64, CancellationToken)` with application layer code can be called or if not, `DisposeAsync()` will use the code provided in `DefaultCloseErrorCode`. Either way, `DisposeAsync()` must be called at the end of the work with the connection to fully release all the associated resources.

Consider the following `QuicConnection` example code:

```

C#

using System.Net.Quic;

// First, check if QUIC is supported.
if (!QuicConnection.IsSupported)
{
    Console.WriteLine("QUIC is not supported, check for presence of libmsquic and support of
TLS 1.3.");
    return;
}

// This represents the minimal configuration necessary to open a connection.
var clientConnectionOptions = new QuicClientConnectionOptions
{
    // End point of the server to connect to.
    RemoteEndPoint = listener.LocalEndPoint,
}

```

```

// Used to abort stream if it's not properly closed by the user.
// See https://www.rfc-editor.org/rfc/rfc9000#section-20.2
DefaultStreamErrorCode = 0x0A, // Protocol-dependent error code.

// Used to close the connection if it's not done by the user.
// See https://www.rfc-editor.org/rfc/rfc9000#section-20.2
DefaultCloseErrorCode = 0x0B, // Protocol-dependent error code.

// Optionally set limits for inbound streams.
MaxInboundUnidirectionalStreams = 10,
MaxInboundBidirectionalStreams = 100,

// Same options as for client side SslStream.
ClientAuthenticationOptions = new SslClientAuthenticationOptions
{
    // List of supported application protocols.
    ApplicationProtocols = new List<SslApplicationProtocol>() { "protocol-name" }
}
};

// Initialize, configure and connect to the server.
var connection = await QuicConnection.ConnectAsync(clientConnectionOptions);

Console.WriteLine($"Connected {connection.LocalEndPoint} --> {connection.RemoteEndPoint}");

// Open a bidirectional (can both read and write) outbound stream.
var outgoingStream = await connection.OpenOutboundStreamAsync(QuicStreamType.Bidirectional);

// Work with the outgoing stream ...

// To accept any stream on a client connection, at least one of
// MaxInboundBidirectionalStreams or MaxInboundUnidirectionalStreams of QuicConnectionOptions
// must be set.
while (isRunning)
{
    // Accept an inbound stream.
    var incomingStream = await connection.AcceptInboundStreamAsync();

    // Work with the incoming stream ...
}

// Close the connection with the custom code.
await connection.CloseAsync(0x0C);

// Dispose the connection.
await connection.DisposeAsync();

```

or more information about how the `QuicConnection` was designed, see the [API proposal ↗](#).

QuicStream

[QuicStream](#) is the actual type that is used to send and receive data in QUIC protocol. It derives from ordinary [Stream](#) and can be used as such, but it also offers several features that are specific to QUIC protocol. Firstly, a QUIC stream can either be unidirectional or bidirectional, see [RFC 9000 Section 2.1 ↗](#). A bidirectional stream is able to send and receive data on both sides, whereas unidirectional stream can only write from the initiating side and read on the accepting one. Each peer can limit how many concurrent stream of each type is willing to accept, see [MaxInboundBidirectionalStreams](#) and [MaxInboundUnidirectionalStreams](#).

Another particularity of QUIC stream is ability to explicitly close the writing side in the middle of work with the stream, see [CompleteWrites\(\)](#) or [WriteAsync\(ReadOnlyMemory<Byte>, Boolean, CancellationToken\)](#) overload with `completeWrites` argument. Closing of the writing side lets the peer know that no more data will arrive, yet the peer still can continue sending (in case of a bidirectional stream). This is useful in scenarios like HTTP request/response exchange when the client sends the request and closes the writing side to let the server know that this is the end of the request content. Server is still able to send the response after that, but knows that no more data will arrive from the client. And for erroneous cases, either writing or reading side of the stream can be aborted, see [Abort\(QuicAbortDirection, Int64\)](#). The behavior of the individual methods for each stream type is summarized in the following table (note that both client and server can open and accept streams):

[] [Expand table](#)

| Method | Peer opening stream | Peer accepting stream |
|--|--|---|
| <code>CanRead</code> | <code>bidirectional: true</code> <code>unidirectional: false</code> | <code>true</code> |
| <code>CanWrite</code> | <code>true</code> | <code>bidirectional: true</code> <code>unidirectional: false</code> |
| <code>ReadAsync</code> | <code>bidirectional: reads data</code> <code>unidirectional: InvalidOperationException</code> | <code>reads data</code> |
| <code>WriteAsync</code> | <code>sends data => peer read returns the data</code> | <code>bidirectional: sends data => peer read returns the data</code> <code>unidirectional: InvalidOperationException</code> |
| <code>CompleteWrites</code> | <code>closes writing side => peer read returns 0</code> | <code>bidirectional: closes writing side => peer read returns 0</code> <code>unidirectional: no-op</code> |
| <code>Abort(QuicAbortDirection.Read)</code> | <code>bidirectional: STOP_SENDING ↗ => peer write throws</code> <code>QuicException(QuicError.OperationAborted)</code> <code>unidirectional: no-op</code> | <code>STOP_SENDING ↗ => peer write throws</code> <code>QuicException(QuicError.OperationAborted)</code> |
| <code>Abort(QuicAbortDirection.Write)</code> | <code>RESET_STREAM ↗ => peer read throws</code> <code>QuicException(QuicError.OperationAborted)</code> | <code>bidirectional: RESET_STREAM ↗ => peer read throws</code> <code>QuicException(QuicError.OperationAborted)</code> <code>unidirectional: no-op</code> |

On top of these methods, `QuicStream` offers two specialized properties to get notified whenever either reading or writing side of the stream has been closed: `ReadsClosed` and `WritesClosed`. Both return a `Task` that completes with its corresponding side getting closed, whether it be success or abort, in which case the `Task` will contain appropriate exception. These properties are useful when the user code needs to know about stream side getting closed without issuing call to `ReadAsync` or `WriteAsync`.

Finally, when the work with the stream is done, it needs to be disposed with [DisposeAsync\(\)](#). The dispose will make sure that both reading and/or writing side - depending on the stream type - is closed. If stream hasn't been properly read till the end, dispose will issue an equivalent of [Abort\(QuicAbortDirection.Read\)](#). However, if stream writing side hasn't been closed, it will be gracefully closed as it would be with `CompleteWrites`. The reason for this difference is to make sure that scenarios working with an ordinary `Stream` behave as expected and lead to a successful path. Consider the following example:

C#

```
// Work done with all different types of streams.
async Task WorkWithStreamAsync(Stream stream)
{
    // This will dispose the stream at the end of the scope.
    await using (stream)
    {
        // Simple echo, read data and send them back.
        byte[] buffer = new byte[1024];
        int count = 0;
        // The loop stops when read returns 0 bytes as is common for all streams.
        while ((count = await stream.ReadAsync(buffer)) > 0)
        {
            await stream.WriteAsync(buffer.AsMemory(0, count));
        }
    }
}

// Open a QuicStream and pass to the common method.
var quicStream = await connection.OpenOutboundStreamAsync(QuicStreamType.Bidirectional);
await WorkWithStreamAsync(quicStream);
```

The sample usage of `QuicStream` in client scenario:

C#

```
// Consider connection from the connection example, open a bidirectional stream.
await using var stream = await connection.OpenStreamAsync(QuicStreamType.Bidirectional,
cancellationToken);

// Send some data.
await stream.WriteAsync(data, cancellationToken);
await stream.WriteAsync(data, cancellationToken);

// End the writing-side together with the last data.
await stream.WriteAsync(data, endStream: true, cancellationToken);
// Or separately.
stream.CompleteWrites();

// Read data until the end of stream.
while (await stream.ReadAsync(buffer, cancellationToken) > 0)
{
    // Handle buffer data...
}

// DisposeAsync called by await using at the top.
```

And the sample usage of `QuicStream` in server scenario:

C#

```
// Consider connection from the connection example, accept a stream.
await using var stream = await connection.AcceptStreamAsync(cancellationToken);

if (stream.Type != QuicStreamType.Bidirectional)
{
    Console.WriteLine($"Expected bidirectional stream, got {stream.Type}");
    return;
}
```

```

// Read the data.
while (stream.ReadAsync(buffer, cancellationToken) > 0)
{
    // Handle buffer data...

    // Client completed the writes, the loop might be exited now without another ReadAsync.
    if (stream.ReadsCompleted.IsCompleted)
    {
        break;
    }
}

// Listen for Abort(QuicAbortDirection.Read) from the client.
var writesClosedTask = WritesClosedAsync(stream);
async ValueTask WritesClosedAsync(QuicStream stream)
{
    try
    {
        await stream.WritesClosed;
    }
    catch (Exception ex)
    {
        // Handle peer aborting our writing side ...
    }
}

// DisposeAsync called by await using at the top.

```

For more information about how the `QuicStream` was designed, see the [API proposal](#).

See also

- [Networking in .NET](#)
- [HTTP/3 with HttpClient](#)
- [System.Net.Quic](#)
- [QuicListener](#)
- [QuicConnection](#)
- [QuicStream](#)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

.NET is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

Troubleshoot QUIC issues in .NET

Article • 05/22/2023

In this article, you'll learn how to diagnose the most common issues with QUIC in .NET.

The `System.Net.Quic` library is based on the open source QUIC implementation [MsQuic](#). Because of this, the behavior differs from ordinary sockets, sometimes by design. Moreover, it's based on UDP protocol, and it doesn't provide the exact same experience as with TCP.

Listener is running but doesn't receive any data

If a listener is running but never receives data, it might be caused by other process listening on the same port. To verify which process is using which port run:

```
Bash
```

```
sudo ss -tulpw
```

This behavior is by design as `MsQuic` uses `SO_REUSEPORT` to achieve better performance. For more information, see [ListenerStart](#) documentation and the original issue [dotnet/runtime#59382](#).

ⓘ Note

This problem doesn't occur on Windows, where MsQuic will attempt to do a port reservation. This causes the application trying to open second listener on the same port fail to start.

Client receives unexpected ALPN error

Client attempts to connect, but receives `Application layer protocol negotiation error was encountered` despite using the same ALPN as the server.

What happens is that listener always binds to dual-mode wildcard address, regardless of what the application specified. Then it matches incoming connections by IP address and ALPN. If no match is found, it reports the above mentioned error. As a result, mismatch between listening IP address and connecting one will result in an ALPN error.

To avoid this error, make sure you're connecting to the same address for which the listener was started. For example, print the listening address for your listener:

```
C#
```

```
await using var listener = QuicListener.ListenAsync(new() /* appropriate
options */);
Console.WriteLine(listener.LocalEndPoint);
```

This issue might happen in several different scenarios, such as in this issue [dotnet/runtime#85412](#). The server was started for `Loopback` address (`127.0.0.1`) and everything worked when ran on the same machine. But when the client tried to connect from a different one, the address would not match the server loopback address and was rejected with ALPN error.

ⓘ Note

This happens when the listener is using MsQuic, for example via .NET `QuicListener`.

Listener succeeds to start for IPv6 despite it being disabled

Despite IPv6 being disabled, `QuicListener.ListenAsync` succeeds with an IPv6 address. This is related to the previous problem as `MsQuic` binds to wildcard address and thus succeeds in doing so. As a result, the listener is started but no connection can be made to it. This is behavioral difference from sockets, which throw in such case.

There are many ways to check if IPv6 is enable, for example on Linux check the state of IPv6 module:

```
Bash
```

```
cat /sys/module/ipv6/parameters/disable

# 0 - IPv6 is enable
# 1 - IPv6 is disabled
```

As stated above, MsQuic behavior here is intentional. However, this particular problem might be mitigated on the .NET side in the future. For more details, see [dotnet/runtime#75343](#).

Listener fails to start with QUIC_STATUS_ADDRESS_IN_USE error

This a Windows specific problem. Listener throws `QuicException` with `QUIC_STATUS_ADDRESS_IN_USE` error despite no other process running on particular address and port. The error is caused by port exclusion range defined for the same port as the listener is trying to listen on. To check for the exclusion ranges run:

```
batch
```

```
netsh.exe int ip show excludedportrange protocol=udp
```

Getting failure for attempt to bind on a port that is from excluded range is expected behavior. For that reason, there are no immediate plans to fix this. More details can be found in [dotnet/runtime#71518](#).

See also

- [MsQuic troubleshooting guide](#)

 Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

Networking telemetry in .NET

Article • 11/30/2023

The .NET networking stack is instrumented at various layers. .NET gives you the option to collect accurate timings throughout the lifetime of an HTTP request using metrics, event counters, and events.

- **Networking metrics:** Starting with .NET 8, the HTTP and the name resolution (DNS) components are instrumented using the modern [System.Diagnostics.Metrics API](#). These metrics were designed in cooperation with [OpenTelemetry](#).
- **Networking events:** Events provide debug and trace information with accurate timestamps.
- **Networking event counters:** All networking components are instrumented to publish real-time performance metrics using the EventCounters API.

💡 Tip

If you're looking for information on tracking HTTP operations across different services, see the [distributed tracing documentation](#).

🌐 Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

.NET is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

Networking metrics in .NET

Article • 11/30/2023

Metrics are numerical measurements reported over time. They are typically used to monitor the health of an app and generate alerts.

Starting with .NET 8, the `System.Net.Http` and the `System.Net.NameResolution` components are instrumented to publish metrics using .NET's new `System.Diagnostics.Metrics API`. These metrics were designed in cooperation with [OpenTelemetry](#) to make sure they're consistent with the standard and work well with popular tools like [Prometheus](#) and [Grafana](#). They are also [multi-dimensional](#), meaning that measurements are associated with key-value pairs called tags (a.k.a. attributes or labels) that allow data to be categorized for analysis.

💡 Tip

For a comprehensive list of all built-in instruments together with their attributes, see [System.Net metrics](#).

Collect System.Net metrics

There are two parts to using metrics in a .NET app:

- **Instrumentation:** Code in .NET libraries takes measurements and associates these measurements with a metric name. .NET and ASP.NET Core include many built-in metrics.
- **Collection:** A .NET app configures named metrics to be transmitted from the app for external storage and analysis. Some tools might perform configuration outside the app using configuration files or a UI tool.

This section demonstrates various methods to collect and view System.Net metrics.

Example app

For the sake of this tutorial, create a simple app that sends HTTP requests to various endpoints in parallel.

.NET CLI

```
dotnet new console -o HelloBuiltinMetrics
```

```
cd ..\HelloBuiltinMetrics
```

Replace the contents of `Program.cs` with the following sample code:

C#

```
using System.Net;

string[] uris = ["http://example.com", "http://httpbin.org/get",
"https://example.com", "https://httpbin.org/get"];
using HttpClient client = new()
{
    DefaultRequestVersion = HttpVersion.Version20
};

Console.WriteLine("Press any key to start.");
Console.ReadKey();

while (!Console.KeyAvailable)
{
    await Parallel.ForAsync(0, Random.Shared.Next(20), async (_, ct) =>
    {
        string uri = uris[Random.Shared.Next(uris.Length)];
        byte[] bytes = await client.GetByteArrayAsync(uri, ct);
        await Console.Out.WriteLineAsync($"{uri} - received {bytes.Length}
bytes.");
    });
}
```

View metrics with dotnet-counters

[dotnet-counters](#) is a cross-platform performance monitoring tool for ad-hoc health monitoring and first-level performance investigation.

.NET CLI

```
dotnet tool install --global dotnet-counters
```

When running against a .NET 8+ process, `dotnet-counters` enables the instruments defined by the `--counters` argument and displays the measurements. It continuously refreshes the console with the latest numbers:

Console

```
dotnet-counters monitor --counters System.Net.Http,System.Net.NameResolution
-n HelloBuiltinMetrics
```

View metrics in Grafana with OpenTelemetry and Prometheus

Overview

[OpenTelemetry](#):

- Is a vendor-neutral, open-source project supported by the [Cloud Native Computing Foundation](#).
- Standardizes generating and collecting telemetry for cloud-native software.
- Works with .NET using the .NET metric APIs.
- Is endorsed by [Azure Monitor](#) and many APM vendors.

This tutorial shows one of the integrations available for OpenTelemetry metrics using the OSS [Prometheus](#) and [Grafana](#) projects. The metrics data flow consists of the following steps:

1. The .NET metric APIs record measurements from the example app.
2. The OpenTelemetry library running in the app aggregates the measurements.
3. The Prometheus exporter library makes the aggregated data available via an HTTP metrics endpoint. 'Exporter' is what OpenTelemetry calls the libraries that transmit telemetry to vendor-specific backends.
4. A Prometheus server:
 - Polls the metrics endpoint.
 - Reads the data.
 - Stores the data in a database for long-term persistence. Prometheus refers to reading and storing data as *scraping* an endpoint.
 - Can run on a different machine.
5. The Grafana server:
 - Queries the data stored in Prometheus and displays it on a web-based monitoring dashboard.
 - Can run on a different machine.

Configure the example app to use OpenTelemetry's Prometheus exporter

Add a reference to the OpenTelemetry Prometheus exporter to the example app:

.NET CLI

```
dotnet add package OpenTelemetry.Exporter.Prometheus.HttpListener --  
prerelease
```

ⓘ Note

This tutorial uses a pre-release build of OpenTelemetry's Prometheus support available at the time of writing.

Update `Program.cs` with OpenTelemetry configuration:

C#

```
using OpenTelemetry.Metrics;  
using OpenTelemetry;  
using System.Net;  
  
using MeterProvider meterProvider = Sdk.CreateMeterProviderBuilder()  
    .AddMeter("System.Net.Http", "System.Net.NameResolution")  
    .AddPrometheusHttpListener(options => options.UriPrefixes = new string[]  
{ "http://localhost:9184/" })  
    .Build();  
  
string[] uris = ["http://example.com", "http://httpbin.org/get",  
"https://example.com", "https://httpbin.org/get"];  
using HttpClient client = new()  
{  
    DefaultRequestVersion = HttpVersion.Version20  
};  
  
while (!Console.KeyAvailable)  
{  
    await Parallel.ForAsync(0, Random.Shared.Next(20), async (_, ct) =>  
    {  
        string uri = uris[Random.Shared.Next(uris.Length)];  
        byte[] bytes = await client.GetByteArrayAsync(uri, ct);  
        await Console.Out.WriteLineAsync($"{uri} - received {bytes.Length}  
bytes.");  
    });  
}
```

In the preceding code:

- `AddMeter("System.Net.Http", "System.Net.NameResolution")` configures OpenTelemetry to transmit all the metrics collected by the built-in `System.Net.Http` and `System.Net.NameResolution` meters.

- `AddPrometheusHttpListener` configures OpenTelemetry to expose Prometheus' metrics HTTP endpoint on port `9184`.

ⓘ Note

This configuration differs for ASP.NET Core apps, where metrics are exported with `OpenTelemetry.Exporter.Prometheus.AspNetCore` instead of `HttpListener`. See the [related ASP.NET Core example](#).

Run the app and leave it running so measurements can be collected:

.NET CLI

```
dotnet run
```

Set up and configure Prometheus

Follow the [Prometheus first steps](#) to set up a Prometheus server and confirm it is working.

Modify the `prometheus.yml` configuration file so that Prometheus scrapes the metrics endpoint that the example app is exposing. Add the following highlighted text in the `scrape_configs` section:

YAML

```
# my global config
global:
  scrape_interval: 15s # Set the scrape interval to every 15 seconds.
  # Default is every 1 minute.
  evaluation_interval: 15s # Evaluate rules every 15 seconds. The default is
  # every 1 minute.
  # scrape_timeout is set to the global default (10s).

# Alertmanager configuration
alerting:
  alertmanagers:
    - static_configs:
        - targets:
            # - alertmanager:9093

# Load rules once and periodically evaluate them according to the global
# 'evaluation_interval'.
rule_files:
  # - "first_rules.yml"
  # - "second_rules.yml"
```

```

# A scrape configuration containing exactly one endpoint to scrape:
# Here it's Prometheus itself.
scrape_configs:
  # The job name is added as a label `job=<job_name>` to any timeseries
  # scraped from this config.
  - job_name: "prometheus"

    # metrics_path defaults to '/metrics'
    # scheme defaults to 'http'.

  static_configs:
    - targets: ["localhost:9090"]

  - job_name: 'OpenTelemetryTest'
    scrape_interval: 1s # poll very quickly for a more responsive demo
    static_configs:
      - targets: ['localhost:9184']

```

Start prometheus

1. Reload the configuration or restart the Prometheus server.
2. Confirm that OpenTelemetryTest is in the UP state in the **Status > Targets** page of the Prometheus web portal.

The screenshot shows the Prometheus web interface with the following details:

- Header:** Prometheus Time Series Collector
- Toolbar:** Includes navigation icons, search bar, and user info (Guest).
- Navigation:** Prometheus > Status > Targets
- Status Selection:** The "Status" dropdown menu is highlighted with a yellow box.
- Targets Table:**

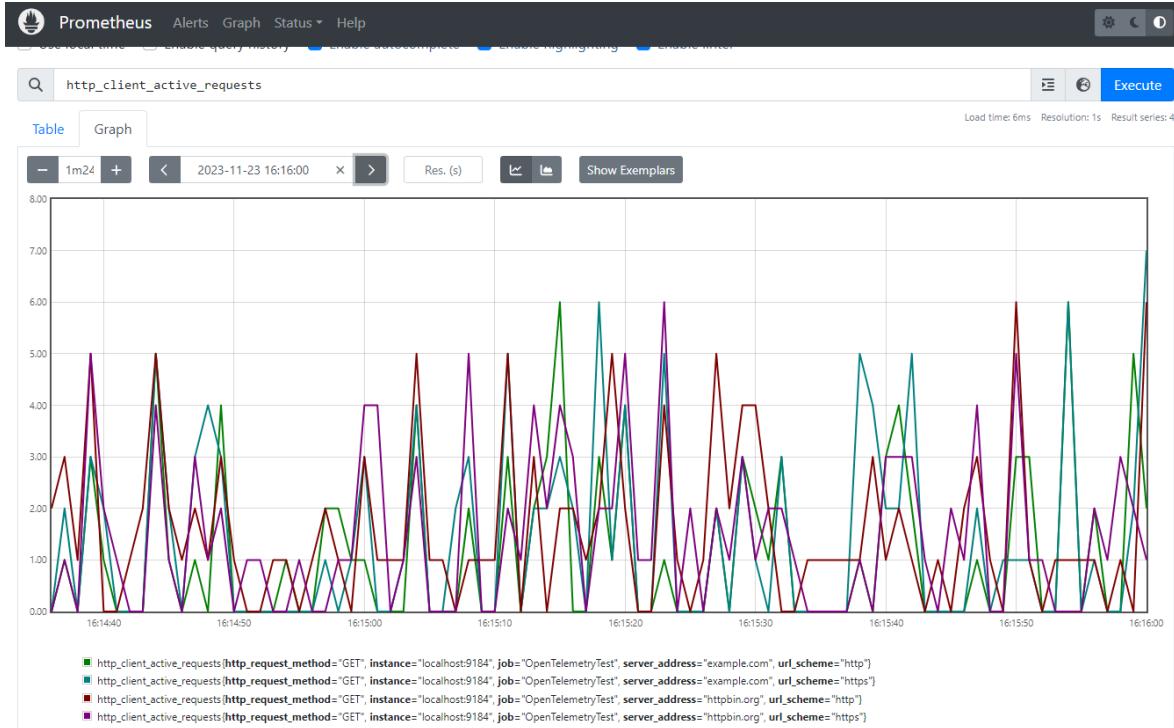
| Endpoint | State | Labels | Last Scrape | Scrape Duration | Error |
|-------------------------------|-------|--|----------------|-----------------|-------|
| http://localhost:9184/metrics | UP | instance="localhost:9184" job="OpenTelemetryTest" | 488.000 ms ago | 0.977ms | |
- prometheus Table:**

| Endpoint | State | Labels | Last Scrape | Scrape Duration | Error |
|-------------------------------|-------|---|-------------|-----------------|-------|
| http://localhost:9090/metrics | UP | instance="localhost:9090" job="prometheus" | 1.921s ago | 10.583ms | |

3. On the Graph page of the Prometheus web portal, enter `http` in the expression text box and select `http_client_active_requests`.

The screenshot shows the Prometheus web interface. At the top, there are several checkboxes: 'Use local time', 'Enable query history', 'Enable autocomplete' (which is checked), 'Enable highlighting' (which is checked), and 'Enable linter'. Below the search bar, a table lists metrics starting with 'http': `http_client_active_requests`, `http_client_open_connections`, `http_client_request_duration_seconds_bucket`, `http_client_request_duration_seconds_count`, and `http_client_request_duration_seconds_sum`. The first metric is highlighted. To its right, a detailed description box says: 'Number of outbound HTTP requests that are currently active on the client.' Below the table, the status bar indicates 'Load time: 6ms Resolution: 1s Result series: 8'.

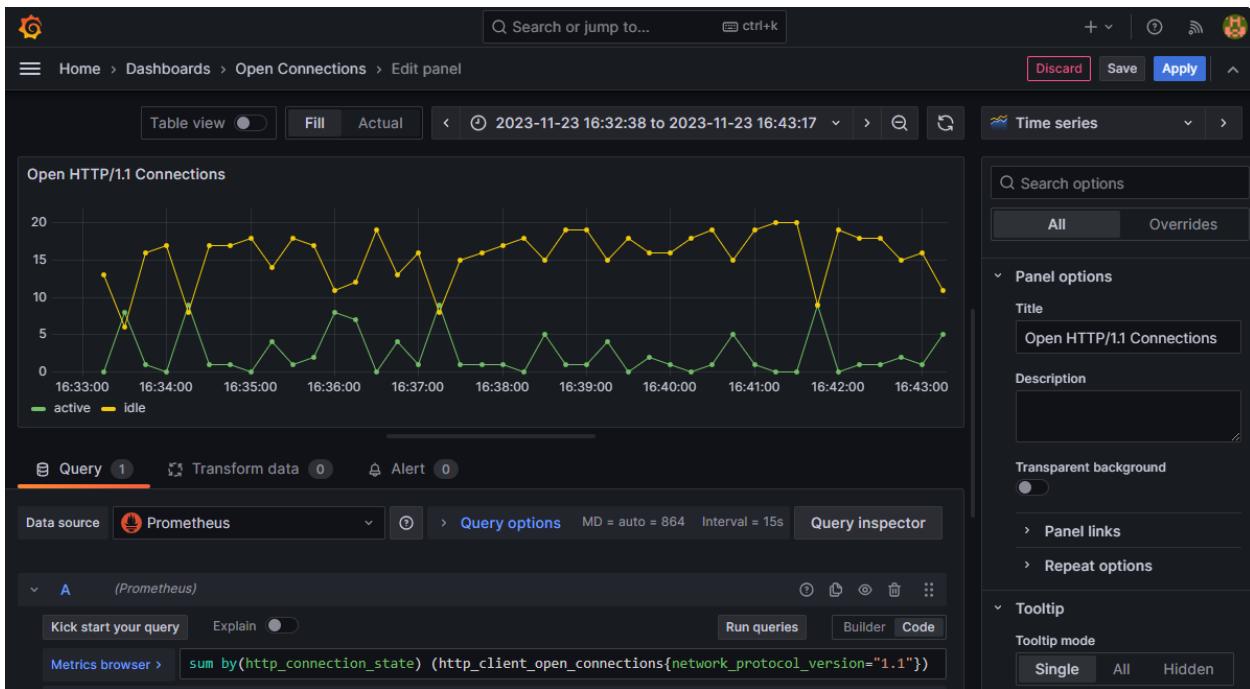
In the graph tab, Prometheus shows the value of the `http.client.active_requests` counter that's emitted by the example app.



Show metrics on a Grafana dashboard

- Follow the [standard instructions](#) to install Grafana and connect it to a Prometheus data source.
- Create a Grafana dashboard by selecting the + icon on the top toolbar then selecting **Dashboard**. In the dashboard editor that appears, enter **Open HTTP/1.1 Connections** in the **Title** box and the following query in the PromQL expression field:

```
sum by(http_connection_state)
(http_client_open_connections{network_protocol_version="1.1"})
```



1. Select **Apply** to save and view the new dashboard. It displays the number of active vs idle HTTP/1.1 connections in the pool.

Enrichment

Enrichment is the addition of custom tags (a.k.a. attributes or labels) to a metric. This is useful if an app wants to add a custom categorization to dashboards or alerts built with metrics. The `http.client.request.duration` instrument supports enrichment by registering callbacks with the `HttpMetricsEnrichmentContext`. Note that this is a low-level API and a separate callback registration is needed for each `HttpRequestMessage`.

A simple way to do the callback registration at a single place is to implement a custom `DelegatingHandler`. This will allow you to intercept and modify the requests before they are forwarded to the inner handler and sent to the server:

C#

```
using System.Net.Http.Metrics;

using HttpClient client = new(new EnrichmentHandler() { InnerHandler = new
HttpClientHandler() });

await client.GetStringAsync("https://httpbin.org/response-headers?
Enrichment-Value=A");
await client.GetStringAsync("https://httpbin.org/response-headers?
Enrichment-Value=B");

sealed class EnrichmentHandler : DelegatingHandler
{
    protected override Task<HttpResponseMessage>
```

```

SendAsync(HttpRequestMessage request, CancellationToken cancellationToken)
{
    HttpMetricsEnrichmentContext.AddCallback(request, static context =>
    {
        if (context.Response is not null) // Response is null when an
        exception occurs.
        {
            // Use any information available on the request or the
            response to emit custom tags.
            string? value =
context.Response.Headers.GetValues("Enrichment-Value").FirstOrDefault();
            if (value != null)
            {
                context.AddCustomTag("enrichment_value", value);
            }
        }
    });
    return base.SendAsync(request, cancellationToken);
}
}

```

If you're working with [IHttpClientFactory](#), you can use [AddHttpMessageHandler](#) to register the `EnrichmentHandler`:

C#

```

using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Options;
using System.Net.Http.Metrics;

ServiceCollection services = new();
services.AddHttpClient(Options.DefaultName).AddHttpMessageHandler(() => new
EnrichmentHandler());

ServiceProvider serviceProvider = services.BuildServiceProvider();
HttpClient client = serviceProvider.GetRequiredService<HttpClient>();

await client.GetStringAsync("https://httpbin.org/response-headers?
Enrichment-Value=A");
await client.GetStringAsync("https://httpbin.org/response-headers?
Enrichment-Value=B");

```

ⓘ Note

For performance reasons, the enrichment callback is only invoked when the `http.client.request.duration` instrument is enabled, meaning that something should be collecting the metrics. This can be `dotnet-monitor`, Prometheus exporter, a `MeterListener`, or a `MetricCollector<T>`.

IMeterFactory and **IHttpClientFactory** integration

HTTP metrics were designed with isolation and testability in mind. These aspects are supported by the use of [IMeterFactory](#), which enables publishing metrics by a custom [Meter](#) instance in order to keep Meters isolated from each other. By default, all metrics are emitted by a global [Meter](#) internal to the [System.Net.Http](#) library. This behavior can be overridden by assigning a custom [IMeterFactory](#) instance to [SocketsHttpHandler.MeterFactory](#) or [HttpClientHandler.MeterFactory](#).

ⓘ Note

The [Meter.Name](#) is [System.Net.Http](#) for all metrics emitted by [HttpClientHandler](#) and [SocketsHttpHandler](#).

When working with [Microsoft.Extensions.Http](#) and [IHttpClientFactory](#) on .NET 8+, the default [IHttpClientFactory](#) implementation automatically picks the [IMeterFactory](#) instance registered in the [IServiceCollection](#) and assigns it to the primary handler it creates internally.

ⓘ Note

Starting with .NET 8, the [AddHttpClient](#) method automatically calls [AddMetrics](#) to initialize the metrics services and register the default [IMeterFactory](#) implementation with [IServiceCollection](#). The default [IMeterFactory](#) caches [Meter](#) instances by name, meaning that there will be one [Meter](#) with the name [System.Net.Http](#) per [IServiceCollection](#).

Test metrics

The following example demonstrates how to validate built-in metrics in unit tests using [xUnit](#), [IHttpClientFactory](#), and [MetricCollector<T>](#) from the [Microsoft.Extensions.Diagnostics.Testing](#) NuGet package:

C#

```
[Fact]
public async Task RequestDurationTest()
{
    // Arrange
```

```

ServiceCollection services = new();
services.AddHttpClient();
ServiceProvider serviceProvider = services.BuildServiceProvider();
var meterFactory = serviceProvider.GetService<IMeterFactory>();
var collector = new MetricCollector<double>(meterFactory,
    "System.Net.Http", "http.client.request.duration");
var client = serviceProvider.GetRequiredService<HttpClient>();

// Act
await client.GetStringAsync("http://example.com");

// Assert
await collector.WaitForMeasurementsAsync(minCount:
1).WaitAsync(TimeSpan.FromSeconds(5));
Assert.Collection(collector.GetMeasurementSnapshot(),
    measurement =>
{
    Assert.Equal("http", measurement.Tags["url.scheme"]);
    Assert.Equal("GET", measurement.Tags["http.request.method"]);
});
}

```

Metrics vs. EventCounters

Metrics are [more feature-rich](#) than EventCounters, most notably because of their multi-dimensional nature. This multi-dimensionality lets you create sophisticated queries in tools like Prometheus and get insights on a level that's not possible with EventCounters.

Nevertheless, as of .NET 8, only the `System.Net.Http` and the `System.Net.NameResolutions` components are instrumented using Metrics, meaning that if you need counters from the lower levels of the stack such as `System.Net.Sockets` or `System.Net.Security`, you must use EventCounters.

Moreover, there are some semantical differences between Metrics and their matching EventCounters. For example, when using `HttpCompletionOption.ResponseContentRead`, the [current-requests](#) EventCounter considers a request to be active until the moment when the last byte of the request body has been read. Its metrics counterpart `http.client.active_requests` doesn't include the time spent reading the response body when counting the active requests.

Need more metrics?

If you have suggestions for other useful information that could be exposed via metrics, create a [dotnet/runtime issue ↗](#).

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

.NET is an open source project.
Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

Networking events in .NET

Article • 11/30/2023

Events give you access to:

- Accurate timestamps throughout the lifetime of an operation. For example, how long it took to connect to the server and how long it took an HTTP request to receive response headers.
- Debug/trace information that may not be obtainable by other means. For example, what kind of decisions the connection pool made and why.

The instrumentation is based on [EventSource](#), allowing you to collect this information from both inside and outside the process.

Event providers

Networking information is split across the following event providers:

- `System.Net.Http` (`HttpClient` and `SocketsHttpHandler`)
- `System.Net.NameResolution` (`Dns`)
- `System.Net.Security` (`SslStream`)
- `System.Net.Sockets`
- `Microsoft.AspNetCore.Hosting`
- `Microsoft.AspNetCore.Server.Kestrel`

The telemetry has some performance overhead when enabled, so make sure to subscribe only to event providers you're actually interested in.

Consume events in-process

Prefer in-process collection when possible for easier event correlation and analysis.

EventListener

[EventListener](#) is an API that allows you to listen to [EventSource](#) events from within the same process that produced them.

C#

```
using System.Diagnostics.Tracing;
```

```

using var listener = new MyListener();

using var client = new HttpClient();
await client.GetStringAsync("https://httpbin.org/get");

public sealed class MyListener : EventListener
{
    protected override void OnEventSourceCreated(EventSource eventSource)
    {
        if (eventSource.Name == "System.Net.Http")
        {
            EnableEvents(eventSource, EventLevel.Informational);
        }
    }

    protected override void OnEventWritten(EventWrittenEventArgs eventData)
    {
        Console.WriteLine($"{DateTime.UtcNow:ss:fff} {eventData.EventName}:
" +
            string.Join(' ', eventData.PayloadNames!.Zip(eventData.Payload!).Select(pair => $"{{pair.First}}={{pair.Second}}")));
    }
}

```

The preceding code prints output similar to the following:

Output

```

00:598 RequestStart: scheme=https host=httpbin.org port=443
pathAndQuery=/get versionMajor=1 versionMinor=1 versionPolicy=0
01:470 ConnectionEstablished: versionMajor=1 versionMinor=1
01:474 RequestLeftQueue: timeOnQueueMilliseconds=470,6214 versionMajor=1
versionMinor=1
01:476 RequestHeadersStart:
01:477 RequestHeadersStop:
01:592 ResponseHeadersStart:
01:593 ResponseHeadersStop:
01:633 ResponseContentStart:
01:635 ResponseContentStop:
01:635 RequestStop:
01:637 ConnectionClosed: versionMajor=1 versionMinor=1

```

Yarp.Telemetry.Consumption

While the `EventListener` approach outlined above is useful for quick experimentation and debugging, the APIs aren't strongly typed and force you to depend on implementation details of the instrumented library.

To address this, .NET created a library that makes it easy to consume networking events in-process: [Yarp.Telemetry.Consumption](#). While the package is currently maintained as part of the [YARP](#) project, it can be used in any .NET application.

To use it, implement the interfaces and methods (events) that you're interested in:

C#

```
public sealed class MyTelemetryConsumer : IHttpTelemetryConsumer,
INetSecurityTelemetryConsumer
{
    public void OnRequestStart(DateTime timestamp, string scheme, string
host, int port, string pathAndQuery, int versionMajor, int versionMinor,
HttpVersionPolicy versionPolicy)
    {
        Console.WriteLine($"Request to {host} started at {timestamp}");
    }

    public void OnHandshakeStart(DateTime timestamp, bool isServer, string
targetHost)
    {
        Console.WriteLine($"Starting TLS handshake with {targetHost}");
    }
}
```

And register the implementations with your DI container:

C#

```
services.AddTelemetryConsumer<MyTelemetryConsumer>();
```

The library provides the following strongly typed interfaces:

- [IHttpTelemetryConsumer](#)
- [INameResolutionTelemetryConsumer](#)
- [INetSecurityTelemetryConsumer](#)
- [ISocketsTelemetryConsumer](#)
- [IKestrelTelemetryConsumer](#)

These callbacks are called as part of the instrumented operation, so the general logging guidance applies. You should avoid blocking or performing any expensive calculations as part of the callback. Offload any post-processing work to different threads to avoid adding latency to the underlying operation.

Consume events from outside the process

dotnet-trace

[dotnet-trace](#) is a cross-platform tool that enables the collection of .NET Core traces of a running process without a native profiler.

```
Console
```

```
dotnet tool install --global dotnet-trace
```

```
Console
```

```
dotnet-trace collect --providers  
System.Net.Http, System.Net.Security, System.Threading.Tasks.TplEventSource:0x  
80:4 --process-id 1234
```

For all the available commands and parameters, see the [dotnet-trace docs](#).

You can analyze the captured `.nettrace` file in Visual Studio or [PerfView](#). For more information, see the [dotnet-trace analysis docs](#).

PerfView

[PerfView](#) is a free, advanced performance analysis tool. It runs on Windows but can also analyze traces captured on Linux.

To configure the list of events to capture, specify them under `Advanced Options / Additional Providers`:

```
txt
```

```
*System.Net.Sockets, *System.Net.NameResolution, *System.Net.Http, *System.Net.  
Security
```

TraceEvent

[TraceEvent](#) is a library that allows you to consume events from different processes in real time. `dotnet-trace` and `PerfView` both rely on it.

If you want to process events programmatically and in real time, see the [TraceEvent docs](#).

Start and Stop events

Larger operations often start with a `Start` event and end with a `Stop` event. For example, you'll see `RequestStart`/`RequestStop` events from `System.Net.Http` or `ConnectStart`/`ConnectStop` events from `System.Net.Sockets`.

While large operations such as these often guarantee that a `Stop` event will always be present, this is not always the case. For example, seeing the `RequestHeadersStart` event from `System.Net.Http` does not guarantee that `RequestHeadersStop` will also be logged.

Event correlation

Now that you know how to observe these events, you often need to correlate them together, generally to the originating HTTP request.

Prefer in-process collection when possible for easier event correlation and analysis.

In-process correlation

As networking uses asynchronous I/O, you can't assume that the thread that completed a given request is also the thread that started it. This means you can't use `[ThreadLocal]` statics to correlate events, but you can use an `AsyncLocal`. Get familiar with `AsyncLocal` as this type is key to correlating work across different threads.

`AsyncLocal` allows you to access the same state deeper into the async flow of an operation. `AsyncLocal` values only flow down (deeper into the async call stack), and don't propagate changes up to the caller.

Consider the following example:

C#

```
AsyncLocal<int> asyncLocal = new();
asyncLocal.Value = 1;

await WorkAsync();
Console.WriteLine($"Value after WorkAsync: {asyncLocal.Value}");

async Task WorkAsync()
{
    Console.WriteLine($"Value in WorkAsync: {asyncLocal.Value}");
    asyncLocal.Value = 2;
    Console.WriteLine($"Value in WorkAsync: {asyncLocal.Value}");
}
```

This code produces the following output:

Output

```
Value in WorkAsync: 1
Value in WorkAsync: 2
Value after WorkAsync: 1
```

The value 1 flowed down from the caller into `WorkAsync`, but the modification in `WorkAsync` (2) did not flow back up to the caller.

As telemetry events (and their corresponding callbacks) occur as part of the underlying operation, they happen within the same async scope as the caller that initiated the request. This means that you can observe `asyncLocal.Value` from within the callback, but if you set the value in the callback, nothing will be able to observe it further up the stack.

The following steps show the general pattern.

1. Create a mutable class that can be updated from inside event callbacks.

C#

```
public sealed class RequestInfo
{
    public DateTime StartTime, HeadersSent;
}
```

2. Set the `AsyncLocal.Value` before the main operation so that the state will flow into the operation.

C#

```
private static readonly AsyncLocal<RequestInfo> _requestInfo = new();

public async Task SendRequestAsync(string url)
{
    var info = new RequestInfo();
    _requestInfo.Value = info;

    info.StartTime = DateTime.UtcNow;
    await _client.GetStringAsync(url);
```

3. Inside the event callbacks, check if the shared state is available and update it.

`AsyncLocal.Value` will be `null` if the request was sent by a component that didn't set the `AsyncLocal.Value` in the first place.

C#

```
public void OnRequestHeadersStop(DateTime timestamp)
{
    if (_requestInfo.Value is { } info) info.HeadersSent = timestamp;
}
```

4. Process the collected information after finishing the operation.

C#

```
await _client.GetStringAsync(url);

Log($"Time until headers were sent {url} was {info.HeadersSent - info.StartTime}");
```

For more ways to do this, see the [samples section](#).

Correlation outside the process

Every event has a piece of data attached to it called `ActivityID`. This ID encodes the async hierarchy at the time the event was produced.

If you look at a trace file in PerfView, you'll see events like:

txt

| | |
|---------------------------------------|-----------------------------|
| System.Net.Http/Request/Start | ActivityID="/#14396/1/1/" |
| System.Net.Http/RequestHeaders/Start | ActivityID="/#14396/1/1/2/" |
| System.Net.Http/RequestHeaders/Stop | ActivityID="/#14396/1/1/2/" |
| System.Net.Http/ResponseHeaders/Start | ActivityID="/#14396/1/1/3/" |
| System.Net.Http/ResponseHeaders/Stop | ActivityID="/#14396/1/1/3/" |
| System.Net.Http/Request/Stop | ActivityID="/#14396/1/1/" |

You'll know that these events belong to the same request because they share the `/#14396/1/1/` prefix.

When doing manual investigations, a useful trick is to look for the `System.Net.Http/Request/Start` event of a request you're interested in, then paste its `ActivityID` in the `Text Filter` text box. If you now select all available providers, you'll see the list of all events that were produced as part of that request.

PerfView automatically collects the `ActivityID`, but if you use tools like `dotnet-trace`, you must explicitly enable the `System.Threading.Tasks.TplEventSource:0x80:4` provider (see `dotnet-trace` example above).

HttpClient request vs. connection lifetime

Since .NET 6, an HTTP request is no longer tied to a specific connection. Instead, the request will be serviced as soon as any connection is available.

This means you may see the following order of events:

1. Request start
2. Dns start
3. Request stop
4. Dns stop

This indicates that the request did trigger a DNS resolution, but was processed by a different connection before the DNS call completed. The same goes for socket connects or TLS handshakes - the originating request may complete before they do.

You should think about such events separately. Monitor DNS resolutions or TLS handshakes on their own without tying them to the timeline of a specific request.

Internal diagnostics

Some components in .NET are instrumented with additional debug-level events that provide more insight into exactly what's happening internally. These events come with high performance overhead and their shape is constantly changing. As the name suggests, they are not part of the public API and you should therefore not rely on their behavior or existence.

Regardless, these events can offer a lot of insights when all else fails. The `System.Net` stack emits such events from `Private.InternalDiagnostics.System.Net.*` namespaces.

If you change the condition in the `EventListener` example above to
`eventSource.Name.Contains("System.Net")`, you will see 100+ events from different layers in the stack. For more information, see the [full example ↗](#).

Samples

- [Measure DNS resolutions for a given endpoint](#)
- [Measure time-to-headers when using HttpClient](#)
- [Time to process a request in ASP.NET Core running Kestrel](#)
- [Measure the latency of a .NET reverse proxy](#)

Measure DNS resolutions for a given endpoint

C#

```
services.AddTelemetryConsumer(new DnsMonitor("httpbin.org"));

public sealed class DnsMonitor : INameResolutionTelemetryConsumer
{
    private static readonly AsyncLocal<DateTime?> _startTimestamp = new();
    private readonly string _hostname;

    public DnsMonitor(string hostname) => _hostname = hostname;

    public void OnResolutionStart(DateTime timestamp, string hostNameOrAddress)
    {
        if (hostNameOrAddress.Equals(_hostname,
StringComparison.OrdinalIgnoreCase))
        {
            _startTimestamp.Value = timestamp;
        }
    }

    public void OnResolutionStop(DateTime timestamp)
    {
        if (_startTimestamp.Value is { } start)
        {
            Console.WriteLine($"DNS resolution for {_hostname} took
{((timestamp - start).TotalMilliseconds} ms");
        }
    }
}
```

Measure time-to-headers when using HttpClient

C#

```
var info = RequestState.Current; // Initialize the AsyncLocal's value ahead
of time

var response = await client.GetStringAsync("http://httpbin.org/get");

var requestTime = (info.RequestStop - info.RequestStart).TotalMilliseconds;
var serverLatency = (info.HeadersReceived -
info.HeadersSent).TotalMilliseconds;
Console.WriteLine($"Request took {requestTime:N2} ms, server request latency
was {serverLatency:N2} ms");

public sealed class RequestState
{
    private static readonly AsyncLocal<RequestState> _asyncLocal = new();
```

```

    public static RequestState Current => _asyncLocal.Value ??= new();

    public DateTime RequestStart;
    public DateTime HeadersSent;
    public DateTime HeadersReceived;
    public DateTime RequestStop;
}

public sealed class TelemetryConsumer : IHttpTelemetryConsumer
{
    public void OnRequestStart(DateTime timestamp, string scheme, string
host, int port, string pathAndQuery, int versionMajor, int versionMinor,
HttpVersionPolicy versionPolicy) =>
        RequestState.Current.RequestStart = timestamp;

    public void OnRequestStop(DateTime timestamp) =>
        RequestState.Current.RequestStop = timestamp;

    public void OnRequestHeadersStop(DateTime timestamp) =>
        RequestState.Current.HeadersSent = timestamp;

    public void OnResponseHeadersStop(DateTime timestamp) =>
        RequestState.Current.HeadersReceived = timestamp;
}

```

Time to process a request in ASP.NET Core running Kestrel

This is currently the most accurate way to measure the duration of a given request.

```

C#

public sealed class KestrelTelemetryConsumer : IKestrelTelemetryConsumer
{
    private static readonly AsyncLocal<DateTime?> _startTimestamp = new();
    private readonly ILogger<KestrelTelemetryConsumer> _logger;

    public KestrelTelemetryConsumer(ILogger<KestrelTelemetryConsumer>
logger) => _logger = logger;

    public void OnRequestStart(DateTime timestamp, string connectionId,
string requestId, string httpVersion, string path, string method)
    {
        _startTimestamp.Value = timestamp;
    }

    public void OnRequestStop(DateTime timestamp, string connectionId,
string requestId, string httpVersion, string path, string method)
    {
        if (_startTimestamp.Value is { } startTime)
        {

```

```
        var elapsed = timestamp - startTime;
        _logger.LogInformation("Request {requestId} to {path} took
{elapsedMs} ms", requestId, path, elapsed.TotalMilliseconds);
    }
}
```

Measure the latency of a .NET reverse proxy

This sample is applicable if you have a reverse proxy that receives inbound requests via Kestrel and makes outbound requests via HttpClient (for example, [YARP](#)).

This sample measures the time from receiving the request headers until they're sent out to the backend server.

C#

```
public sealed class InternalLatencyMonitor : IKestrelTelemetryConsumer,
IHttpTelemetryConsumer
{
    private record RequestInfo(DateTime StartTimestamp, string RequestId,
string Path);

    private static readonly AsyncLocal<RequestInfo> _requestInfo = new();
    private readonly ILogger<InternalLatencyMonitor> _logger;

    public InternalLatencyMonitor(ILogger<InternalLatencyMonitor> logger) =>
        _logger = logger;

    public void OnRequestStart(DateTime timestamp, string connectionId,
string requestId, string httpVersion, string path, string method)
    {
        _requestInfo.Value = new RequestInfo(timestamp, requestId, path);
    }

    public void OnRequestHeadersStop(DateTime timestamp)
    {
        if (_requestInfo.Value is { } requestInfo)
        {
            var elapsed = (timestamp -
requestInfo.StartTimestamp).TotalMilliseconds;
            _logger.LogInformation("Internal latency for {requestId} to
{path} was {duration} ms", requestInfo.RequestId, requestInfo.Path,
elapsed);
        }
    }
}
```

Need more telemetry?

If you have suggestions for other useful information that could be exposed via events or metrics, create a [dotnet/runtime issue ↗](#).

If you're using the [Yarp.Telemetry.Consumption ↗](#) library and have any suggestions, create a [microsoft/reverse-proxy issue ↗](#).

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

.NET is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

Networking event counters in .NET

Article • 11/30/2023

[EventCounters](#) are .NET APIs used for lightweight, cross-platform, and near real-time performance metric collection.

Networking components are instrumented to publish basic diagnostic information using EventCounters. They include information like the following:

- `System.Net.Http > requests-started`
- `System.Net.Http > requests-failed`
- `System.Net.Http > http11-connections-current-total`
- `System.Net.Security > all-tls-sessions-open`
- `System.Net.Sockets > outgoing-connections-established`
- `System.Net.NameResolution > dns-lookups-duration`

💡 Tip

For the full list, see [well-known counters](#).

💡 Tip

On projects targeting .NET 8+, consider using the newer and more feature-rich [networking metrics](#) instead of EventCounters.

Providers

Networking information is split across the following providers:

- `System.Net.Http` (`HttpClient` and `SocketsHttpHandler`)
- `System.Net.NameResolution` (`Dns`)
- `System.Net.Security` (`SslStream`)
- `System.Net.Sockets`
- `Microsoft.AspNetCore.Hosting`
- `Microsoft.AspNetCore.Server.Kestrel`

The telemetry has some performance overhead when enabled, so make sure to subscribe only to providers you're actually interested in.

Monitor event counters from outside the process

dotnet-counters

[dotnet-counters](#) is a cross-platform performance monitoring tool for ad-hoc health monitoring and first-level performance investigation.

.NET CLI

```
dotnet tool install --global dotnet-counters
```

.NET CLI

```
dotnet-counters monitor --counters System.Net.Http,System.Net.Security --process-id 1234
```

The command continually refreshes the console with the latest numbers.

txt

| [System.Net.Http] | |
|---------------------------------------|-----|
| Current Http 1.1 Connections | 3 |
| Current Http 2.0 Connections | 1 |
| Current Http 3.0 Connections | 0 |
| Current Requests | 4 |
| HTTP 1.1 Requests Queue Duration (ms) | 0 |
| HTTP 2.0 Requests Queue Duration (ms) | 0 |
| HTTP 3.0 Requests Queue Duration (ms) | 0 |
| Requests Failed | 0 |
| Requests Failed Rate (Count / 1 sec) | 0 |
| Requests Started | 470 |
| Requests Started Rate (Count / 1 sec) | 18 |

For all the available commands and parameters, see the [dotnet-counter docs](#).

Application Insights

Application Insights does not collect event counters by default. For information on customizing the set of counters you're interested in, see the [AppInsights EventCounters docs](#).

For example:

C#

```
services.ConfigureTelemetryModule<EventCounterCollectionModule>((module,
options) =>
{
    module.Counters.Add(new EventCounterCollectionRequest("System.Net.Http",
"current-requests"));
    module.Counters.Add(new EventCounterCollectionRequest("System.Net.Http",
"requests-failed"));
    module.Counters.Add(new EventCounterCollectionRequest("System.Net.Http",
"http11-connections-current-total"));
    module.Counters.Add(new
EventCounterCollectionRequest("System.Net.Security", "all-tls-sessions-
open"));
});
```

For an example of how to subscribe to many runtime and ASP.NET event counters, see the [RuntimeEventCounters sample](#). Simply add an `EventCounterCollectionRequest` for every entry.

C#

```
foreach (var (eventSource, counters) in RuntimeEventCounters.EventCounters)
{
    foreach (string counter in counters)
    {
        module.Counters.Add(new EventCounterCollectionRequest(eventSource,
counter));
    }
}
```

Consume event counters in-process

The [Yarp.Telemetry.Consumption](#) library makes it easy to consume event counters from within the process. While the package is currently maintained as part of the [YARP](#) project, it can be used in any .NET application.

To use it, implement the `IMetricsConsumer<TMetrics>` interface:

C#

```
public sealed class MyMetricsConsumer : IMetricsConsumer<SocketsMetrics>
{
    public void OnMetrics(SocketsMetrics previous, SocketsMetrics current)
    {
        var elapsedTime = (current.Timestamp -
previous.Timestamp).TotalMilliseconds;
        Console.WriteLine($"Received {current.BytesReceived -
```

```
previous.BytesReceived} bytes in the last {elapsedTime:N2} ms");
    }
}
```

Then register the implementations with your DI container:

C#

```
services.AddSingleton<IMetricsConsumer<SocketsMetrics>, MyMetricsConsumer>();
services.AddTelemetryListeners();
```

The library provides the following strongly typed metrics types:

- [HttpMetrics ↗](#)
- [NameResolutionMetrics ↗](#)
- [NetSecurityMetrics ↗](#)
- [SocketsMetrics ↗](#)
- [KestrelMetrics ↗](#)

Need more telemetry?

If you have suggestions for other useful information that could be exposed via events or metrics, create a [dotnet/runtime issue ↗](#).

If you're using the [Yarp.Telemetry.Consumption ↗](#) library and have any suggestions, create a [microsoft/reverse-proxy issue ↗](#).

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

.NET is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

File globbing in .NET

Article • 03/18/2023

In this article, you'll learn how to use file globbing with the [Microsoft.Extensions.FileSystemGlobbing](#) NuGet package. A *glob* is a term used to define patterns for matching file and directory names based on wildcards. Globbing is the act of defining one or more glob patterns, and yielding files from either inclusive or exclusive matches.

Patterns

To match files in the file system based on user-defined patterns, start by instantiating a `Matcher` object. A `Matcher` can be instantiated with no parameters, or with a `System.StringComparison` parameter, which is used internally for comparing patterns to file names. The `Matcher` exposes the following additive methods:

- `Matcher.AddExclude`
- `Matcher.AddInclude`

Both `AddExclude` and `AddInclude` methods can be called any number of times, to add various file name patterns to either exclude or include from results. Once you've instantiated a `Matcher` and added patterns, it's then used to evaluate matches from a starting directory with the `Matcher.Execute` method.

Extension methods

The `Matcher` object has several extension methods.

Multiple exclusions

To add multiple exclude patterns, you can use:

C#

```
Matcher matcher = new();
matcher.AddExclude("*.txt");
matcher.AddExclude("*.asciidoc");
matcher.AddExclude("*.md");
```

Alternatively, you can use the [MatcherExtensions.AddExcludePatterns\(Matcher, IEnumerable<String>\[\]\)](#) to add multiple exclude patterns in a single call:

C#

```
Matcher matcher = new();
matcher.AddExcludePatterns(new [] { "*.txt", "*.asciidoc", "*.md" });
```

This extension method iterates over all of the provided patterns calling [AddExclude](#) on your behalf.

Multiple inclusions

To add multiple include patterns, you can use:

C#

```
Matcher matcher = new();
matcher.AddInclude("*.txt");
matcher.AddInclude("*.asciidoc");
matcher.AddInclude("*.md");
```

Alternatively, you can use the [MatcherExtensions.AddIncludePatterns\(Matcher, IEnumerable<String>\[\]\)](#) to add multiple include patterns in a single call:

C#

```
Matcher matcher = new();
matcher.AddIncludePatterns(new[] { "*.txt", "*.asciidoc", "*.md" });
```

This extension method iterates over all of the provided patterns calling [AddInclude](#) on your behalf.

Get all matching files

To get all matching files, you have to call [Matcher.Execute\(DirectoryInfoBase\)](#) either directly or indirectly. To call it directly, you need a search directory:

C#

```
Matcher matcher = new();
matcher.AddIncludePatterns(new[] { "*.txt", "*.asciidoc", "*.md" });

string searchDirectory = "../starting-folder/";
```

```

PatternMatchingResult result = matcher.Execute(
    new DirectoryInfoWrapper(
        new DirectoryInfo(searchDirectory)));

// Use result.HasMatches and results.Files.
// The files in the results object are file paths relative to the search
directory.

```

The preceding C# code:

- Instantiates a `Matcher` object.
- Calls `AddIncludePatterns(Matcher, IEnumerable<String>[])` to add several file name patterns to include.
- Declares and assigns the search directory value.
- Instantiates a `DirectoryInfo` from the given `searchDirectory`.
- Instantiates a `DirectoryInfoWrapper` from the `DirectoryInfo` it wraps.
- Calls `Execute` given the `DirectoryInfoWrapper` instance to yield a `PatternMatchingResult` object.

ⓘ Note

The `DirectoryInfoWrapper` type is defined in the `Microsoft.Extensions.FileSystemGlobbing.Abstractions` namespace, and the `DirectoryInfo` type is defined in the `System.IO` namespace. To avoid unnecessary `using` statements, you can use the provided extension methods.

There is another extension method that yields an `IEnumerable<string>` representing the matching files:

C#

```

Matcher matcher = new();
matcher.AddIncludePatterns(new[] { ".txt", ".asciidoc", ".md" });

string searchDirectory = "../starting-folder/";

IEnumerable<string> matchingFiles =
    matcher.GetResultsInFullPath(searchDirectory);

// Use matchingFiles if there are any found.
// The files in this collection are fully qualified file system paths.

```

The preceding C# code:

- Instantiates a `Matcher` object.

- Calls `AddIncludePatterns(Matcher, IEnumerable<String>[])` to add several file name patterns to include.
- Declares and assigns the search directory value.
- Calls `GetResultsInFullPath` given the `searchDirectory` value to yield all matching files as a `IEnumerable<string>`.

Match overloads

The `PatternMatchingResult` object represents a collection of `FilePatternMatch` instances, and exposes a `boolean` value indicating whether the result has matches—`PatternMatchingResult.HasMatches`.

With a `Matcher` instance, you can call any of the various `Match` overloads to get a pattern matching result. The `Match` methods invert the responsibility on the caller to provide a file or a collection of files in which to evaluate for matches. In other words, the caller is responsible for passing the file to match on.

ⓘ Important

When using any of the `Match` overloads, there is no file system I/O involved. All of the file globbing is done in memory with the include and exclude patterns of the `matcher` instance. The parameters of the `Match` overloads do not have to be fully qualified paths. The current directory (`Directory.GetCurrentDirectory()`) is used when not specified.

To match a single file:

```
C#
Matcher matcher = new();
matcher.AddInclude("/**/*.md");

PatternMatchingResult result = matcher.Match("file.md");
```

The preceding C# code:

- Matches any file with the `.md` file extension, at an arbitrary directory depth.
- If a file named `file.md` exists in a subdirectory from the current directory:
 - `result.HasMatches` would be `true`.
 - and `result.Files` would have one match.

The additional `Match` overloads work in similar ways.

Pattern formats

The patterns that are specified in the `AddExclude` and `AddInclude` methods can use the following formats to match multiple files or directories.

- Exact directory or file name
 - `some-file.txt`
 - `path/to/file.txt`
- Wildcards `*` in file and directory names that represent zero to many characters not including separator characters.

| Value | Description |
|---------------------------|---|
| <code>*.txt</code> | All files with <code>.txt</code> file extension. |
| <code>*.*</code> | All files with an extension. |
| <code>*</code> | All files in top-level directory. |
| <code>.*</code> | File names beginning with <code>'.'</code> . |
| <code>*word*</code> | All files with 'word' in the filename. |
| <code>readme.*</code> | All files named 'readme' with any file extension. |
| <code>styles/*.css</code> | All files with extension <code>'.css'</code> in the directory <code>'styles/'</code> . |
| <code>scripts/**/*</code> | All files in <code>'scripts/'</code> or one level of subdirectory under <code>'scripts/'</code> . |
| <code>images*/*</code> | All files in a folder with name that is or begins with <code>'images'</code> . |

- Arbitrary directory depth (`/**/`).

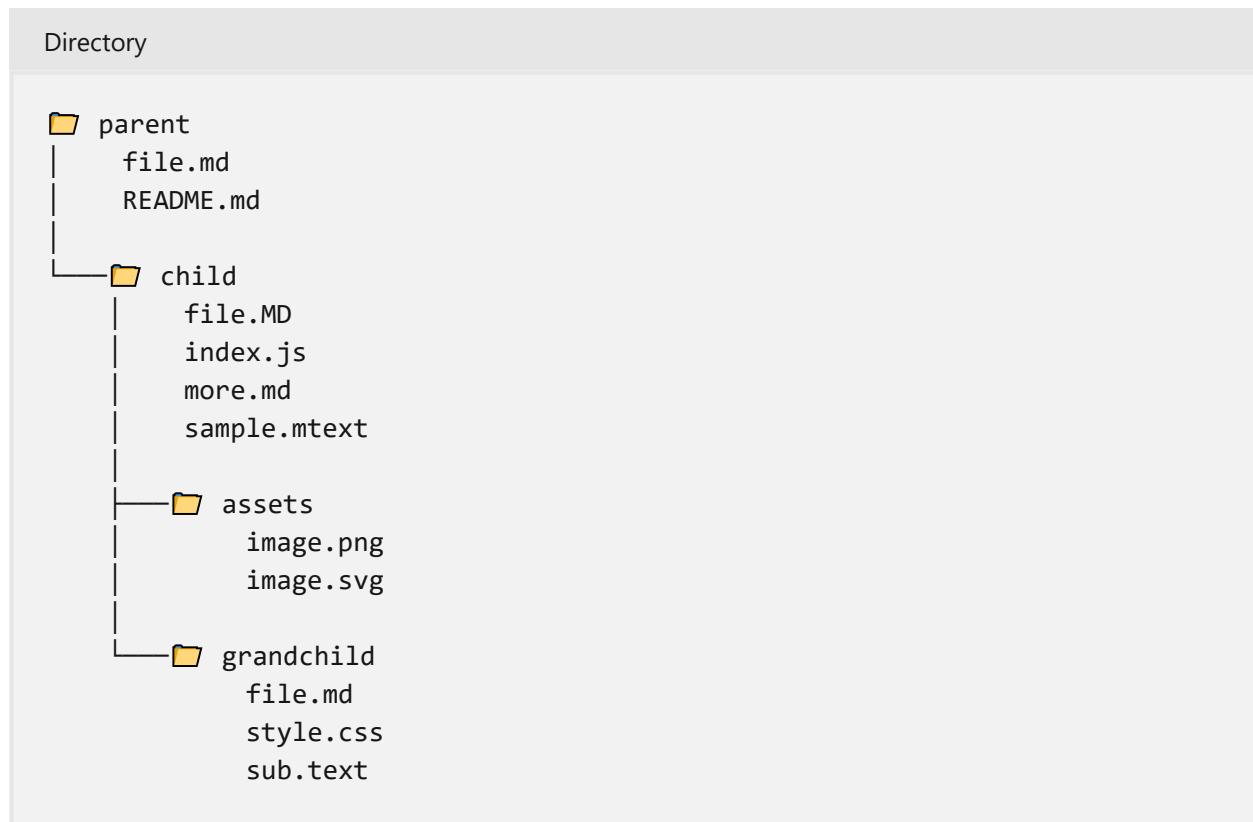
| Value | Description |
|-----------------------|---|
| <code>**/*</code> | All files in any subdirectory. |
| <code>dir/</code> | All files in any subdirectory under <code>'dir/'</code> . |
| <code>dir/**/*</code> | All files in any subdirectory under <code>'dir/'</code> . |

- Relative paths.

To match all files in a directory named "shared" at the sibling level to the base directory given to `Matcher.Execute(DirectoryInfoBase)`, use `../shared/*`.

Examples

Consider the following example directory, and each file within its corresponding folder.



💡 Tip

Some file extensions are in uppercase, while others are in lowercase. By default, **StringComparer.OrdinalIgnoreCase** is used. To specify different string comparison behavior, use the **Matcher.Matcher(StringComparison)** constructor.

To get all of the markdown files, where the file extension is either `.md` or `.mtext`, regardless of character case:

C#

```
Matcher matcher = new();
matcher.AddIncludePatterns(new[] { "**/*.md", "**/*.mtext" });

foreach (string file in matcher.GetResultsInFullPath("parent"))
{
    Console.WriteLine(file);
}
```

Running the application would output results similar to the following:

Console

```
C:\app\parent\file.md
C:\app\parent\README.md
C:\app\parent\child\file.MD
C:\app\parent\child\more.md
C:\app\parent\child\sample.mtext
C:\app\parent\child\grandchild\file.md
```

To get any files in an *assets* directory at an arbitrary depth:

C#

```
Matcher matcher = new();
matcher.AddInclude("**/assets/**/*");

foreach (string file in matcher.GetResultsInFullPath("parent"))
{
    Console.WriteLine(file);
}
```

Running the application would output results similar to the following:

Console

```
C:\app\parent\child\assets\image.png
C:\app\parent\child\assets\image.svg
```

To get any files where the directory name contains the word *child* at an arbitrary depth, and the file extensions are not *.md*, *.text*, or *.mtext*:

C#

```
Matcher matcher = new();
matcher.AddInclude("**/*child/**/*");
matcher.AddExcludePatterns(
    new[]
    {
        "**/*.md", "**/*.text", "**/*.mtext"
    });

foreach (string file in matcher.GetResultsInFullPath("parent"))
{
    Console.WriteLine(file);
}
```

Running the application would output results similar to the following:

Console

```
C:\app\parent\child\index.js  
C:\app\parent\child\assets\image.png  
C:\app\parent\child\assets\image.svg  
C:\app\parent\child\grandchild\style.css
```

See also

- [Runtime libraries overview](#)
- [File and Stream I/O](#)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback [here](#).

 [Open a documentation issue](#)

 [Provide product feedback](#)

Primitives: The extensions library for .NET

Article • 03/18/2023

In this article, you'll learn about the [Microsoft.Extensions.Primitives](#) library. The primitives in this article are *not* to be confused with .NET primitive types from the BCL, or that of the C# language. Instead, the types within the primitive's library serve as building blocks for some of the peripheral .NET NuGet packages, such as:

- [Microsoft.Extensions.Configuration](#)
- [Microsoft.Extensions.Configuration.FileExtensions](#)
- [Microsoft.Extensions.FileProviders.Composite](#)
- [Microsoft.Extensions.FileProviders.Physical](#)
- [Microsoft.Extensions.Logging.EventSource](#)
- [Microsoft.Extensions.Options](#)
- [System.Text.Json](#)

Change notifications

Propagating notifications when a change occurs is a fundamental concept in programming. The observed state of an object more often than not can change. When change occurs, implementations of the [Microsoft.Extensions.Primitives.IChangeToken](#) interface can be used to notify interested parties of said change. The implementations available are as follows:

- [CancellationChangeToken](#)
- [CompositeChangeToken](#)

As a developer, you're also free to implement your own type. The [IChangeToken](#) interface defines a few properties:

- [IChangeToken.HasChanged](#): Gets a value that indicates if a change has occurred.
- [IChangeToken.ActiveChangeCallbacks](#): Indicates if the token will proactively raise callbacks. If `false`, the token consumer must poll `HasChanged` to detect changes.

Instance-based functionality

Consider the following example usage of the `CancellationChangeToken`:

C#

```

CancellationTokenSource cancellationTokenSource = new();
CancellationChangeToken cancellationChangeToken =
new(cancellationTokenSource.Token);

Console.WriteLine($"HasChanged: {cancellationChangeToken.HasChanged}");

static void callback(object? _) =>
    Console.WriteLine("The callback was invoked.");

using (IDisposable subscription =
    cancellationChangeToken.RegisterChangeCallback(callback, null))
{
    cancellationTokenSource.Cancel();
}

Console.WriteLine($"HasChanged: {cancellationChangeToken.HasChanged}\n");

// Outputs:
//     HasChanged: False
//     The callback was invoked.
//     HasChanged: True

```

In the preceding example, a [CancellationTokenSource](#) is instantiated and its [Token](#) is passed to the [CancellationChangeToken](#) constructor. The initial state of [HasChanged](#) is written to the console. An [Action<object?>](#) [callback](#) is created that writes when the callback is invoked to the console. The token's [RegisterChangeCallback\(Action<Object>, Object\)](#) method is called, given the [callback](#). Within the [using](#) statement, the [cancellationTokenSource](#) is cancelled. This triggers the callback, and the state of [HasChanged](#) is again written to the console.

When you need to take action from multiple sources of change, use the [CompositeChangeToken](#). This implementation aggregates one or more change tokens and fires each registered callback exactly one time regardless of the number of times a change is triggered. Consider the following example:

C#

```

CancellationTokenSource firstCancellationTokenSource = new();
CancellationChangeToken firstCancellationChangeToken =
new(firstCancellationTokenSource.Token);

CancellationTokenSource secondCancellationTokenSource = new();
CancellationChangeToken secondCancellationChangeToken =
new(secondCancellationTokenSource.Token);

CancellationTokenSource thirdCancellationTokenSource = new();
CancellationChangeToken thirdCancellationChangeToken =
new(thirdCancellationTokenSource.Token);

```

```

var compositeChangeToken =
    new CompositeChangeToken(
        new IChangeToken[]
    {
        firstCancellationChangeToken,
        secondCancellationChangeToken,
        thirdCancellationChangeToken
    });

static void callback(object? state) =>
    Console.WriteLine($"The {state} callback was invoked.");

// 1st, 2nd, 3rd, and 4th.
compositeChangeToken.RegisterChangeCallback(callback, "1st");
compositeChangeToken.RegisterChangeCallback(callback, "2nd");
compositeChangeToken.RegisterChangeCallback(callback, "3rd");
compositeChangeToken.RegisterChangeCallback(callback, "4th");

// It doesn't matter which cancellation source triggers the change.
// If more than one trigger the change, each callback is only fired once.
Random random = new();
int index = random.Next(3);
CancellationTokenSource[] sources = new[]
{
    firstCancellationTokenSource,
    secondCancellationTokenSource,
    thirdCancellationTokenSource
};
sources[index].Cancel();

Console.WriteLine();

// Outputs:
//      The 4th callback was invoked.
//      The 3rd callback was invoked.
//      The 2nd callback was invoked.
//      The 1st callback was invoked.

```

In the preceding C# code, three `CancellationTokenSource` objects instances are created and paired with corresponding `CancellationChangeToken` instances. The composite token is instantiated by passing an array of the tokens to the `CompositeChangeToken` constructor. The `Action<object?> callback` is created, but this time the `state` object is used and written to console as a formatted message. The callback is registered four times, each with a slightly different state object argument. The code uses a pseudo-random number generator to pick one of the change token sources (doesn't matter which one) and call its `Cancel()` method. This triggers the change, invoking each registered callback exactly once.

Alternative static approach

As an alternative to calling `RegisterChangeCallback`, you could use the `Microsoft.Extensions.Primitives.ChangeToken` static class. Consider the following consumption pattern:

C#

```
CancellationTokenSource cancellationTokenSource = new();
CancellationChangeToken cancellationChangeToken =
    new(cancellationTokenSource.Token);

IChangeToken producer()
{
    // The producer factory should always return a new change token.
    // If the token's already fired, get a new token.
    if (cancellationTokenSource.IsCancellationRequested)
    {
        cancellationTokenSource = new();
        cancellationChangeToken = new(cancellationTokenSource.Token);
    }

    return cancellationChangeToken;
}

void consumer() => Console.WriteLine("The callback was invoked.");

using (ChangeToken.OnChange(producer, consumer))
{
    cancellationTokenSource.Cancel();
}

// Outputs:
//      The callback was invoked.
```

Much like previous examples, you'll need an implementation of `IChangeToken` that is produced by the `changeTokenProducer`. The producer is defined as a `Func<IChangeToken>` and it's expected that this will return a new token every invocation. The `consumer` is either an `Action` when not using `state`, or an `Action<TState>` where the generic type `TState` flows through the change notification.

String tokenizers, segments, and values

Interacting with strings is commonplace in application development. Various representations of strings are parsed, split, or iterated over. The primitives library offers

a few choice types that help to make interacting with strings more optimized and efficient. Consider the following types:

- **StringSegment**: An optimized representation of a substring.
- **StringTokenizer**: Tokenizes a `string` into `StringSegment` instances.
- **StringValues**: Represents `null`, zero, one, or many strings in an efficient way.

The `StringSegment` type

In this section, you'll learn about an optimized representation of a substring known as the `StringSegment` `struct` type. Consider the following C# code example showing some of the `StringSegment` properties and the `AsSpan` method:

C#

```
var segment =
    new StringSegment(
        "This a string, within a single segment representation.",
        14, 25);

Console.WriteLine($"Buffer: \'{segment.Buffer}\'");
Console.WriteLine($"Offset: {segment.Offset}");
Console.WriteLine($"Length: {segment.Length}");
Console.WriteLine($"Value: \'{segment.Value}\');

Console.Write("Span: \");
foreach (char @char in segment.AsSpan())
{
    Console.Write(@char);
}
Console.Write("\n");

// Outputs:
//     Buffer: "This a string, within a single segment representation."
//     Offset: 14
//     Length: 25
//     Value: " within a single segment "
//             " within a single segment "
```

The preceding code instantiates the `StringSegment` given a `string` value, an `offset`, and a `length`. The `StringSegment.Buffer` is the original string argument, and the `StringSegment.Value` is the substring based on the `StringSegment.Offset` and `StringSegment.Length` values.

The `StringSegment` struct provides [many methods](#) for interacting with the segment.

The `StringTokenizer` type

The `StringTokenizer` object is a struct type that tokenizes a `string` into `StringSegment` instances. The tokenization of large strings usually involves splitting the string apart and iterating over it. With that said, `String.Split` probably comes to mind. These APIs are similar, but in general, `StringTokenizer` provides better performance. First, consider the following example:

C#

```
var tokenizer =
    new StringTokenizer(
        s_nineHundredAutoGeneratedParagraphsOfLoremIpsum,
        new[] { ' ' });

foreach (StringSegment segment in tokenizer)
{
    // Interact with segment
}
```

In the preceding code, an instance of the `StringTokenizer` type is created given 900 auto-generated paragraphs of Lorem Ipsum text and an array with a single value of a white-space character `' '`. Each value within the tokenizer is represented as a `StringSegment`. The code iterates the segments, allowing the consumer to interact with each `segment`.

Benchmark comparing `StringTokenizer` to `string.Split`

With the various ways of slicing and dicing strings, it feels appropriate to compare two methods with a benchmark. Using the [BenchmarkDotNet](#) NuGet package, consider the following two benchmark methods:

1. Using `StringTokenizer`:

C#

```
StringBuilder buffer = new();

var tokenizer =
    new StringTokenizer(
        s_nineHundredAutoGeneratedParagraphsOfLoremIpsum,
        new[] { ' ', '.' });

foreach (StringSegment segment in tokenizer)
{
```

```
        buffer.Append(segment.Value);  
    }
```

2. Using `String.Split`:

C#

```
StringBuilder buffer = new();  
  
string[] tokenizer =  
    s_nineHundredAutoGeneratedParagraphsOfLoremIpsum.Split(  
        new[] { ' ', '.' });  
  
foreach (string segment in tokenizer)  
{  
    buffer.Append(segment);  
}
```

Both methods look similar on the API surface area, and they're both capable of splitting a large string into chunks. The benchmark results below show that the `StringTokenizer` approach is nearly three times faster, but *results may vary*. As with all performance considerations, you should evaluate your specific use case.

| Method | Mean | Error | StdDev | Ratio |
|-----------|-----------|-----------|-----------|-------|
| Tokenizer | 3.315 ms | 0.0659 ms | 0.0705 ms | 0.32 |
| Split | 10.257 ms | 0.2018 ms | 0.2552 ms | 1.00 |

Legend

- Mean: Arithmetic mean of all measurements
- Error: Half of 99.9% confidence interval
- Standard deviation: Standard deviation of all measurements
- Median: Value separating the higher half of all measurements (50th percentile)
- Ratio: Mean of the ratio distribution (Current/Baseline)
- Ratio standard deviation: Standard deviation of the ratio distribution (Current/Baseline)
- 1 ms: 1 Millisecond (0.001 sec)

For more information on benchmarking with .NET, see [BenchmarkDotNet](#).

The `StringValues` type

The `StringValues` object is a `struct` type that represents `null`, zero, one, or many strings in an efficient way. The `StringValues` type can be constructed with either of the following syntaxes: `string?` or `string?[]?`. Using the text from the previous example, consider the following C# code:

C#

```
StringValues values =
    new(s_nineHundredAutoGeneratedParagraphsOfLoremIpsum.Split(
        new[] { '\n' }));

Console.WriteLine($"Count = {values.Count:#,#}");

foreach (string? value in values)
{
    // Interact with the value
}
// Outputs:
//      Count = 1,799
```

The preceding code instantiates a `StringValues` object given an array of string values. The `StringValues.Count` is written to the console.

The `StringValues` type is an implementation of the following collection types:

- `IList<string>`
- `ICollection<string>`
- `IEnumerable<string>`
- `IEnumerable`
- `IReadOnlyList<string>`
- `IReadOnlyCollection<string>`

As such, it can be iterated over and each `value` can be interacted with as needed.

See also

- [Options pattern in .NET](#)
- [Configuration in .NET](#)
- [Logging providers in .NET](#)



Collaborate with us on

.NET

.NET feedback

GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

Globalize and localize .NET applications

Article • 03/18/2023

Developing a world-ready application, including an application that can be localized into one or more languages, involves three steps: globalization, localizability review, and localization.

Globalization

This step involves designing and coding an application that is culture-neutral and language-neutral, and that supports localized user interfaces and regional data for all users. It involves making design and programming decisions that are not based on culture-specific assumptions. While a globalized application is not localized, it nevertheless is designed and written so that it can be subsequently localized into one or more languages with relative ease.

Localizability review

This step involves reviewing an application's code and design to ensure that it can be localized easily and to identify potential roadblocks for localization, and verifying that the application's executable code is separated from its resources. If the globalization stage was effective, the localizability review will confirm the design and coding choices made during globalization. The localizability stage may also identify any remaining issues so that an application's source code doesn't have to be modified during the localization stage.

Localization

This step involves customizing an application for specific cultures or regions. If the globalization and localizability steps have been performed correctly, localization consists primarily of translating the user interface.

Following these three steps provides two advantages:

- It frees you from having to retrofit an application that is designed to support a single culture, such as U.S. English, to support additional cultures.
- It results in localized applications that are more stable and have fewer bugs.

.NET provides extensive support for the development of world-ready and localized applications. In particular, many type members in the .NET class library aid globalization by returning values that reflect the conventions of either the current user's culture or a

specified culture. Also, .NET supports satellite assemblies, which facilitate the process of localizing an application.

In this section

[Globalization](#)

Discusses the first stage of creating a world-ready application, which involves designing and coding an application that is culture-neutral and language-neutral.

[.NET globalization and ICU](#)

Describes how .NET globalization uses [International Components for Unicode \(ICU\)](#).

[Localizability review](#)

Discusses the second stage of creating a localized application, which involves identifying potential roadblocks to localization.

[Localization](#)

Discusses the final stage of creating a localized application, which involves customizing an application's user interface for specific regions or cultures.

[Culture-insensitive string operations](#)

Describes how to use .NET methods and classes that are culture-sensitive by default to obtain culture-insensitive results.

[Best practices for developing world-ready applications](#)

Describes the best practices to follow for globalization, localization, and developing world-ready ASP.NET applications.

Reference

- [System.Globalization](#) namespace

Contains classes that define culture-related information, including the language, the country/region, the calendars in use, the format patterns for dates, currency, and numbers, and the sort order for strings.

- [System.Resources](#) namespace

Provides classes for creating, manipulating, and using resources.

- [System.Text namespace](#)

Contains classes representing ASCII, ANSI, Unicode, and other character encodings.

- [Resgen.exe \(Resource File Generator\)](#)

Describes how to use Resgen.exe to convert .txt files and XML-based resource format (.resx) files to common language runtime binary .resources files.

- [Winres.exe \(Windows Forms Resource Editor\)](#)

Describes how to use Winres.exe to localize Windows Forms forms.

 **Collaborate with us on GitHub**

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

 .NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

Globalization

Article • 03/18/2023

Globalization involves designing and developing a world-ready app that supports localized interfaces and regional data for users in multiple cultures. Before beginning the design phase, you should determine which cultures your app will support. Although an app targets a single culture or region as its default, you can design and write it so that it can easily be extended to users in other cultures or regions.

As developers, we all have assumptions about user interfaces and data that are formed by our cultures. For example, for an English-speaking developer in the United States, serializing date and time data as a string in the format `MM/dd/yyyy hh:mm:ss` seems perfectly reasonable. However, deserializing that string on a system in a different culture is likely to throw a [FormatException](#) exception or produce inaccurate data. Globalization enables us to identify such culture-specific assumptions and ensure that they do not affect our app's design or code.

This article discusses some of the major issues you should consider and the best practices you can follow when handling strings, date and time values, and numeric values in a globalized app.

Strings

The handling of characters and strings is a central focus of globalization, because each culture or region may use different characters and character sets and sort them differently. This section provides recommendations for using strings in globalized apps.

Use Unicode internally

By default, .NET uses Unicode strings. A Unicode string consists of zero, one, or more [Char](#) objects, each of which represents a UTF-16 code unit. There is a Unicode representation for almost every character in every character set in use throughout the world.

Many applications and operating systems, including the Windows operating system, can also use code pages to represent character sets. Code pages typically contain the standard ASCII values from 0x00 through 0x7F and map other characters to the remaining values from 0x80 through 0xFF. The interpretation of values from 0x80 through 0xFF depends on the specific code page. Because of this, you should avoid using code pages in a globalized app if possible.

The following example illustrates the dangers of interpreting code page data when the default code page on a system is different from the code page on which the data was saved. (To simulate this scenario, the example explicitly specifies different code pages.) First, the example defines an array that consists of the uppercase characters of the Greek alphabet. It encodes them into a byte array by using code page 737 (also known as MS-DOS Greek) and saves the byte array to a file. If the file is retrieved and its byte array is decoded by using code page 737, the original characters are restored. However, if the file is retrieved and its byte array is decoded by using code page 1252 (or Windows-1252, which represents characters in the Latin alphabet), the original characters are lost.

C#

```
using System;
using System.IO;
using System.Text;

public class Example
{
    public static void CodePages()
    {
        // Represent Greek uppercase characters in code page 737.
        char[] greekChars =
        {
            'Α', 'Β', 'Γ', 'Δ', 'Ε', 'Ζ', 'Η', 'Θ',
            'Ι', 'Κ', 'Λ', 'Μ', 'Ν', 'Ξ', 'Ο', 'Π',
            'Ρ', 'Σ', 'Τ', 'Υ', 'Φ', 'Χ', 'Ψ', 'Ω'
        };

        Encoding.RegisterProvider(CodePagesEncodingProvider.Instance);

        Encoding cp737 = Encoding.GetEncoding(737);
        int nBytes = cp737.GetByteCount(greekChars);
        byte[] bytes737 = new byte[nBytes];
        bytes737 = cp737.GetBytes(greekChars);
        // Write the bytes to a file.
        FileStream fs = new FileStream(@".\\CodePageBytes.dat",
        FileMode.Create);
        fs.Write(bytes737, 0, bytes737.Length);
        fs.Close();

        // Retrieve the byte data from the file.
        fs = new FileStream(@".\\CodePageBytes.dat", FileMode.Open);
        byte[] bytes1 = new byte[fs.Length];
        fs.Read(bytes1, 0, (int)fs.Length);
        fs.Close();

        // Restore the data on a system whose code page is 737.
        string data = cp737.GetString(bytes1);
        Console.WriteLine(data);
        Console.WriteLine();
```

```
// Restore the data on a system whose code page is 1252.
Encoding cp1252 = Encoding.GetEncoding(1252);
data = cp1252.GetString(bytes1);
Console.WriteLine(data);
}

// The example displays the following output:
//      ΑΒΓΔΕΖΗΘΙΚΛΜΝΞΟΠΡΣΤΥΦΧΨΩ
//      €,ƒ,,...†‡^%Š<ŒŽ‘”•—
```

The use of Unicode ensures that the same code units always map to the same characters, and that the same characters always map to the same byte arrays.

Use resource files

Even if you are developing an app that targets a single culture or region, you should use resource files to store strings and other resources that are displayed in the user interface. You should never add them directly to your code. Using resource files has a number of advantages:

- All the strings are in a single location. You don't have to search throughout your source code to identify strings to modify for a specific language or culture.
- There's no need to duplicate strings. Developers who don't use resource files often define the same string in multiple source code files. This duplication increases the probability that one or more instances will be overlooked when a string is modified.
- You can include non-string resources, such as images or binary data, in the resource file instead of storing them in a separate standalone file, so they can be retrieved easily.

Using resource files has particular advantages if you are creating a localized app. When you deploy resources in satellite assemblies, the common language runtime automatically selects a culture-appropriate resource based on the user's current UI culture as defined by the [CultureInfo.CurrentCulture](#) property. As long as you provide an appropriate culture-specific resource and correctly instantiate a [ResourceManager](#) object or use a strongly typed resource class, the runtime handles the details of retrieving the appropriate resources.

For more information about creating resource files, see [Creating resource files](#). For information about creating and deploying satellite assemblies, see [Create satellite assemblies](#) and [Package and Deploy resources](#).

Search and compare strings

Whenever possible, you should handle strings as entire strings instead of handling them as a series of individual characters. This is especially important when you sort or search for substrings, to prevent problems associated with parsing combined characters.

Tip

You can use the [StringInfo](#) class to work with the text elements rather than the individual characters in a string.

In string searches and comparisons, a common mistake is to treat the string as a collection of characters, each of which is represented by a [Char](#) object. In fact, a single character may be formed by one, two, or more [Char](#) objects. Such characters are found most frequently in strings from cultures whose alphabets consist of characters outside the Unicode Basic Latin character range (U+0021 through U+007E). The following example tries to find the index of the LATIN CAPITAL LETTER A WITH GRAVE character (U+00C0) in a string. However, this character can be represented in two different ways: as a single code unit (U+00C0) or as a composite character (two code units: U+0041 and U+0300). In this case, the character is represented in the string instance by two [Char](#) objects, U+0041 and U+0300. The example code calls the [String.IndexOf\(Char\)](#) and [String.IndexOf\(String\)](#) overloads to find the position of this character in the string instance, but these return different results. The first method call has a [Char](#) argument; it performs an ordinal comparison and therefore cannot find a match. The second call has a [String](#) argument; it performs a culture-sensitive comparison and therefore finds a match.

C#

```
using System;
using System.Globalization;
using System.Threading;

public class Example17
{
    public static void Main17()
    {
        Thread.CurrentThread.CurrentCulture =
CultureInfo.CreateSpecificCulture("pl-PL");
        string composite = "\u0041\u0300";
        Console.WriteLine("Comparing using Char: {0}",
composite.IndexOf('\u00C0'));
        Console.WriteLine("Comparing using String: {0}",
composite.IndexOf("\u00C0"));
    }
}
```

```
}
```

```
// The example displays the following output:
//      Comparing using Char: -1
//      Comparing using String: 0
```

You can avoid some of the ambiguity of this example (calls to two similar overloads of a method returning different results) by calling an overload that includes a [StringComparison](#) parameter, such as the [String.IndexOf\(String, StringComparison\)](#) or [String.LastIndexOf\(String, StringComparison\)](#) method.

However, searches are not always culture-sensitive. If the purpose of the search is to make a security decision or to allow or disallow access to some resource, the comparison should be ordinal, as discussed in the next section.

Test strings for equality

If you want to test two strings for equality rather than determine how they compare in the sort order, use the [String.Equals](#) method instead of a string comparison method such as [String.Compare](#) or [CompareInfo.Compare](#).

Comparisons for equality are typically performed to access some resource conditionally. For example, you might perform a comparison for equality to verify a password or to confirm that a file exists. Such non-linguistic comparisons should always be ordinal rather than culture-sensitive. In general, you should call the instance [String.Equals\(String, StringComparison\)](#) method or the static [String.Equals\(String, String, StringComparison\)](#) method with a value of [StringComparison.Ordinal](#) for strings such as passwords, and a value of [StringComparison.OrdinalIgnoreCase](#) for strings such as file names or URIs.

Comparisons for equality sometimes involve searches or substring comparisons rather than calls to the [String.Equals](#) method. In some cases, you may use a substring search to determine whether that substring equals another string. If the purpose of this comparison is non-linguistic, the search should also be ordinal rather than culture-sensitive.

The following example illustrates the danger of a culture-sensitive search on non-linguistic data. The `AccessesFileSystem` method is designed to prohibit file system access for URIs that begin with the substring "FILE". To do this, it performs a culture-sensitive, case-insensitive comparison of the beginning of the URI with the string "FILE". Because a URI that accesses the file system can begin with either "FILE:" or "file:", the implicit assumption is that "i" (U+0069) is always the lowercase equivalent of "I" (U+0049). However, in Turkish and Azerbaijani, the uppercase version of "i" is "İ".

(U+0130). Because of this discrepancy, the culture-sensitive comparison allows file system access when it should be prohibited.

C#

```
using System;
using System.Globalization;
using System.Threading;

public class Example10
{
    public static void Main10()
    {
        Thread.CurrentThread.CurrentCulture =
CultureInfo.CreateSpecificCulture("tr-TR");
        string uri = @"file:\\c:\\users\\username\\Documents\\bio.txt";
        if (!AccessesFileSystem(uri))
            // Permit access to resource specified by URI
            Console.WriteLine("Access is allowed.");
        else
            // Prohibit access.
            Console.WriteLine("Access is not allowed.");
    }

    private static bool AccessesFileSystem(string uri)
    {
        return uri.StartsWith("FILE", true, CultureInfo.CurrentCulture);
    }
}

// The example displays the following output:
//       Access is allowed.
```

You can avoid this problem by performing an ordinal comparison that ignores case, as the following example shows.

C#

```
using System;
using System.Globalization;
using System.Threading;

public class Example11
{
    public static void Main11()
    {
        Thread.CurrentThread.CurrentCulture =
CultureInfo.CreateSpecificCulture("tr-TR");
        string uri = @"file:\\c:\\users\\username\\Documents\\bio.txt";
        if (!AccessesFileSystem(uri))
            // Permit access to resource specified by URI
            Console.WriteLine("Access is allowed.");
```

```

        else
            // Prohibit access.
            Console.WriteLine("Access is not allowed.");
    }

    private static bool AccessesFileSystem(string uri)
    {
        return uri.StartsWith("FILE", StringComparison.OrdinalIgnoreCase);
    }
}

// The example displays the following output:
//       Access is not allowed.

```

Order and sort strings

Typically, ordered strings that are to be displayed in the user interface should be sorted based on culture. For the most part, such string comparisons are handled implicitly by .NET when you call a method that sorts strings, such as [Array.Sort](#) or [List<T>.Sort](#). By default, strings are sorted by using the sorting conventions of the current culture. The following example illustrates the difference when an array of strings is sorted by using the conventions of the English (United States) culture and the Swedish (Sweden) culture.

C#

```

using System;
using System.Globalization;
using System.Threading;

public class Example18
{
    public static void Main18()
    {
        string[] values = { "able", "ångström", "apple", "Æble",
                           "Windows", "Visual Studio" };
        // Change thread to en-US.
        Thread.CurrentThread.CurrentCulture =
CultureInfo.CreateSpecificCulture("en-US");
        // Sort the array and copy it to a new array to preserve the order.
        Array.Sort(values);
        string[] enValues = (String[])values.Clone();

        // Change culture to Swedish (Sweden).
        Thread.CurrentThread.CurrentCulture =
CultureInfo.CreateSpecificCulture("sv-SE");
        Array.Sort(values);
        string[] svValues = (String[])values.Clone();

        // Compare the sorted arrays.
        Console.WriteLine("{0,-8} {1,-15} {2,-15}\n", "Position", "en-US",

```

```

"sv-SE");
    for (int ctr = 0; ctr <= values.GetUpperBound(0); ctr++)
        Console.WriteLine("{0,-8} {1,-15} {2,-15}", ctr, enValues[ctr],
svValues[ctr]);
}
}

// The example displays the following output:
//      Position en-US          sv-SE
//
//      0      able            able
//      1      Äble           Äble
//      2      ångström       apple
//      3      apple          Windows
//      4      Visual Studio  Visual Studio
//      5      Windows         ångström

```

Culture-sensitive string comparison is defined by the [CompareInfo](#) object, which is returned by each culture's [CultureInfo.CompareInfo](#) property. Culture-sensitive string comparisons that use the [String.Compare](#) method overloads also use the [CompareInfo](#) object.

.NET uses tables to perform culture-sensitive sorts on string data. The content of these tables, which contain data on sort weights and string normalization, is determined by the version of the Unicode standard implemented by a particular version of .NET. The following table lists the versions of Unicode implemented by the specified versions of .NET. This list of supported Unicode versions applies to character comparison and sorting only; it does not apply to classification of Unicode characters by category. For more information, see the "Strings and The Unicode Standard" section in the [String](#) article.

| .NET Framework version | Operating system | Unicode version |
|---|-------------------------|------------------------|
| .NET Framework 2.0 | All operating systems | Unicode 4.1 |
| .NET Framework 3.0 | All operating systems | Unicode 4.1 |
| .NET Framework 3.5 | All operating systems | Unicode 4.1 |
| .NET Framework 4 | All operating systems | Unicode 5.0 |
| .NET Framework 4.5 and later on Windows 7 | | Unicode 5.0 |

| .NET Framework version | Operating system | Unicode version |
|---|------------------|--|
| .NET Framework 4.5 and later on Windows 8 and later operating systems | | Unicode 6.3.0 |
| .NET Core and .NET 5+ | | Depends on the version of the Unicode Standard supported by the underlying operating system. |

Starting with .NET Framework 4.5 and in all versions of .NET Core and .NET 5+, string comparison and sorting depends on the operating system. .NET Framework 4.5 and later running on Windows 7 retrieves data from its own tables that implement Unicode 5.0. .NET Framework 4.5 and later running on Windows 8 and later retrieves data from operating system tables that implement Unicode 6.3. On .NET Core and .NET 5+, the supported version of Unicode depends on the underlying operating system. If you serialize culture-sensitive sorted data, you can use the [SortVersion](#) class to determine when your serialized data needs to be sorted so that it is consistent with .NET and the operating system's sort order. For an example, see the [SortVersion](#) class topic.

If your app performs extensive culture-specific sorts of string data, you can work with the [SortKey](#) class to compare strings. A sort key reflects the culture-specific sort weights, including the alphabetic, case, and diacritic weights of a particular string. Because comparisons using sort keys are binary, they are faster than comparisons that use a [CompareInfo](#) object either implicitly or explicitly. You create a culture-specific sort key for a particular string by passing the string to the [CompareInfo.GetSortKey](#) method.

The following example is similar to the previous example. However, instead of calling the [Array.Sort\(Array\)](#) method, which implicitly calls the [CompareInfo.Compare](#) method, it defines an [System.Collections.Generic.IComparer<T>](#) implementation that compares sort keys, which it instantiates and passes to the [Array.Sort<T>\(T\[\], IComparer<T>\)](#) method.

C#

```

using System;
using System.Collections.Generic;
using System.Globalization;
using System.Threading;

public class SortKeyComparer : IComparer<String>
{
    public int Compare(string? str1, string? str2)
    {
        return (str1, str2) switch
    }
}

```

```

        {
            (null, null) => 0,
            (null, _) => -1,
            (_, null) => 1,
            (var s1, var s2) => SortKey.Compare(
                CultureInfo.CurrentCulture.CompareInfo.GetSortKey(s1),
                CultureInfo.CurrentCulture.CompareInfo.GetSortKey(s1))
        };
    }
}

public class Example19
{
    public static void Main19()
    {
        string[] values = { "able", "ångström", "apple", "Æble",
                            "Windows", "Visual Studio" };
        SortKeyComparer comparer = new SortKeyComparer();

        // Change thread to en-US.
        Thread.CurrentThread.CurrentCulture =
CultureInfo.CreateSpecificCulture("en-US");
        // Sort the array and copy it to a new array to preserve the order.
        Array.Sort(values, comparer);
        string[] enValues = (String[])values.Clone();

        // Change culture to Swedish (Sweden).
        Thread.CurrentThread.CurrentCulture =
CultureInfo.CreateSpecificCulture("sv-SE");
        Array.Sort(values, comparer);
        string[] svValues = (String[])values.Clone();

        // Compare the sorted arrays.
        Console.WriteLine("{0,-8} {1,-15} {2,-15}\n", "Position", "en-US",
"sv-SE");
        for (int ctr = 0; ctr <= values.GetUpperBound(0); ctr++)
            Console.WriteLine("{0,-8} {1,-15} {2,-15}", ctr, enValues[ctr],
svValues[ctr]);
    }
}

// The example displays the following output:
//      Position en-US          sv-SE
//
//      0      able      able
//      1      Æble      Æble
//      2      ångström   apple
//      3      apple     Windows
//      4      Visual Studio  Visual Studio
//      5      Windows    ångström

```

Avoid string concatenation

If at all possible, avoid using composite strings that are built at run time from concatenated phrases. Composite strings are difficult to localize, because they often assume a grammatical order in the app's original language that does not apply to other localized languages.

Handle dates and times

How you handle date and time values depends on whether they are displayed in the user interface or persisted. This section examines both usages. It also discusses how you can handle time zone differences and arithmetic operations when working with dates and times.

Display dates and times

Typically, when dates and times are displayed in the user interface, you should use the formatting conventions of the user's culture, which is defined by the [CultureInfo.CurrentCulture](#) property and by the [DateTimeFormatInfo](#) object returned by the [CultureInfo.CurrentCulture.DateTimeFormat](#) property. The formatting conventions of the current culture are automatically used when you format a date by using any of these methods:

- The parameterless [DateTime.ToString\(\)](#) method
- The [DateTime.ToString\(String\)](#) method, which includes a format string
- The parameterless [DateTimeOffset.ToString\(\)](#) method
- The [DateTimeOffset.ToString\(String\)](#), which includes a format string
- The [composite formatting](#) feature, when it is used with dates

The following example displays sunrise and sunset data twice for October 11, 2012. It first sets the current culture to Croatian (Croatia), and then to English (United Kingdom). In each case, the dates and times are displayed in the format that is appropriate for that culture.

C#

```
using System;
using System.Globalization;
using System.Threading;

public class Example3
{
    static DateTime[] dates = { new DateTime(2012, 10, 11, 7, 06, 0),
```

```

        new DateTime(2012, 10, 11, 18, 19, 0) };

    public static void Main3()
    {
        Thread.CurrentThread.CurrentCulture =
CultureInfo.CreateSpecificCulture("hr-HR");
        ShowDayInfo();
        Console.WriteLine();
        Thread.CurrentThread.CurrentCulture =
CultureInfo.CreateSpecificCulture("en-GB");
        ShowDayInfo();
    }

    private static void ShowDayInfo()
    {
        Console.WriteLine("Date: {0:D}", dates[0]);
        Console.WriteLine("    Sunrise: {0:T}", dates[0]);
        Console.WriteLine("    Sunset: {0:T}", dates[1]);
    }
}

// The example displays the following output:
//      Date: 11. listopada 2012.
//          Sunrise: 7:06:00
//          Sunset: 18:19:00
//
//      Date: 11 October 2012
//          Sunrise: 07:06:00
//          Sunset: 18:19:00

```

Persist dates and times

You should never persist date and time data in a format that can vary by culture. This is a common programming error that results in either corrupted data or a run-time exception. The following example serializes two dates, January 9, 2013 and August 18, 2013, as strings by using the formatting conventions of the English (United States) culture. When the data is retrieved and parsed by using the conventions of the English (United States) culture, it is successfully restored. However, when it is retrieved and parsed by using the conventions of the English (United Kingdom) culture, the first date is wrongly interpreted as September 1, and the second fails to parse because the Gregorian calendar does not have an eighteenth month.

C#

```

using System;
using System.IO;
using System.Globalization;
using System.Threading;

```

```
public class Example4
{
    public static void Main4()
    {
        // Persist two dates as strings.
        Thread.CurrentThread.CurrentCulture =
CultureInfo.CreateSpecificCulture("en-US");
        DateTime[] dates = { new DateTime(2013, 1, 9),
                            new DateTime(2013, 8, 18) };
        StreamWriter sw = new StreamWriter("dateData.dat");
        sw.WriteLine("{0:d}|{1:d}", dates[0], dates[1]);
        sw.Close();

        // Read the persisted data.
        StreamReader sr = new StreamReader("dateData.dat");
        string dateData = sr.ReadToEnd();
        sr.Close();
        string[] dateStrings = dateData.Split('|');

        // Restore and display the data using the conventions of the en-US
culture.
        Console.WriteLine("Current Culture: {0}",
                           Thread.CurrentThread.CurrentCulture.DisplayName);
        foreach (var dateStr in dateStrings)
        {
            DateTime restoredDate;
            if (DateTime.TryParse(dateStr, out restoredDate))
                Console.WriteLine("The date is {0:D}", restoredDate);
            else
                Console.WriteLine("ERROR: Unable to parse {0}", dateStr);
        }
        Console.WriteLine();

        // Restore and display the data using the conventions of the en-GB
culture.
        Thread.CurrentThread.CurrentCulture =
CultureInfo.CreateSpecificCulture("en-GB");
        Console.WriteLine("Current Culture: {0}",
                           Thread.CurrentThread.CurrentCulture.DisplayName);
        foreach (var dateStr in dateStrings)
        {
            DateTime restoredDate;
            if (DateTime.TryParse(dateStr, out restoredDate))
                Console.WriteLine("The date is {0:D}", restoredDate);
            else
                Console.WriteLine("ERROR: Unable to parse {0}", dateStr);
        }
    }

    // The example displays the following output:
    //      Current Culture: English (United States)
    //      The date is Wednesday, January 09, 2013
    //      The date is Sunday, August 18, 2013
    //
}
```

```
//      Current Culture: English (United Kingdom)
//      The date is 01 September 2013
//      ERROR: Unable to parse 8/18/2013
```

You can avoid this problem in any of three ways:

- Serialize the date and time in binary format rather than as a string.
- Save and parse the string representation of the date and time by using a custom format string that is the same regardless of the user's culture.
- Save the string by using the formatting conventions of the invariant culture.

The following example illustrates the last approach. It uses the formatting conventions of the invariant culture returned by the static [CultureInfo.InvariantCulture](#) property.

C#

```
using System;
using System.IO;
using System.Globalization;
using System.Threading;

public class Example5
{
    public static void Main5()
    {
        // Persist two dates as strings.
        Thread.CurrentThread.CurrentCulture =
CultureInfo.CreateSpecificCulture("en-US");
        DateTime[] dates = { new DateTime(2013, 1, 9),
                           new DateTime(2013, 8, 18) };
        StreamWriter sw = new StreamWriter("dateData.dat");
        sw.WriteLine(String.Format(CultureInfo.InvariantCulture,
                               "{0:d}|{1:d}", dates[0], dates[1]));
        sw.Close();

        // Read the persisted data.
        StreamReader sr = new StreamReader("dateData.dat");
        string dateData = sr.ReadToEnd();
        sr.Close();
        string[] dateStrings = dateData.Split('|');

        // Restore and display the data using the conventions of the en-US
culture.
        Console.WriteLine("Current Culture: {0}",
                         Thread.CurrentThread.CurrentCulture.DisplayName);
        foreach (var dateStr in dateStrings)
        {
            DateTime restoredDate;
            if (DateTime.TryParse(dateStr, CultureInfo.InvariantCulture,
                                 DateTimeStyles.None, out restoredDate))
                Console.WriteLine("The date is {0:D}", restoredDate);
        }
    }
}
```

```

        else
            Console.WriteLine("ERROR: Unable to parse {0}", dateStr);
    }
    Console.WriteLine();

    // Restore and display the data using the conventions of the en-GB
    culture.
    Thread.CurrentThread.CurrentCulture =
CultureInfo.CreateSpecificCulture("en-GB");
    Console.WriteLine("Current Culture: {0}",
                      Thread.CurrentThread.CurrentCulture.DisplayName);
    foreach (var dateStr in dateStrings)
    {
        DateTime restoredDate;
        if (DateTime.TryParse(dateStr, CultureInfo.InvariantCulture,
                             DateTimeStyles.None, out restoredDate))
            Console.WriteLine("The date is {0:D}", restoredDate);
        else
            Console.WriteLine("ERROR: Unable to parse {0}", dateStr);
    }
}

// The example displays the following output:
//      Current Culture: English (United States)
//      The date is Wednesday, January 09, 2013
//      The date is Sunday, August 18, 2013
//
//      Current Culture: English (United Kingdom)
//      The date is 09 January 2013
//      The date is 18 August 2013

```

Serialization and time zone awareness

A date and time value can have multiple interpretations, ranging from a general time ("The stores open on January 2, 2013, at 9:00 A.M.") to a specific moment in time ("Date of birth: January 2, 2013 6:32:00 A.M."). When a time value represents a specific moment in time and you restore it from a serialized value, you should ensure that it represents the same moment in time regardless of the user's geographical location or time zone.

The following example illustrates this problem. It saves a single local date and time value as a string in three [standard formats](#):

- "G" for general date long time.
- "s" for sortable date/time.
- "o" for round-trip date/time.

```

using System;
using System.IO;

public class Example6
{
    public static void Main6()
    {
        DateTime dateOriginal = new DateTime(2023, 3, 30, 18, 0, 0);
        dateOriginal = DateTime.SpecifyKind(dateOriginal,
DateTimeKind.Local);

        // Serialize a date.
        if (!File.Exists("DateInfo.dat"))
        {
            StreamWriter sw = new StreamWriter("DateInfo.dat");
            sw.WriteLine("{0:G}|{0:s}|{0:o}", dateOriginal);
            sw.Close();
            Console.WriteLine("Serialized dates to DateInfo.dat");
        }
        Console.WriteLine();

        // Restore the date from string values.
        StreamReader sr = new StreamReader("DateInfo.dat");
        string datesToSplit = sr.ReadToEnd();
        string[] dateStrings = datesToSplit.Split('|');
        foreach (var dateStr in dateStrings)
        {
            DateTime newDate = DateTime.Parse(dateStr);
            Console.WriteLine('{0} --> {1} {2}',
                dateStr, newDate, newDate.Kind);
        }
    }
}

```

When the data is restored on a system in the same time zone as the system on which it was serialized, the deserialized date and time values accurately reflect the original value, as the output shows:

Console

```

'3/30/2013 6:00:00 PM' --> 3/30/2013 6:00:00 PM Unspecified
'2013-03-30T18:00:00' --> 3/30/2013 6:00:00 PM Unspecified
'2013-03-30T18:00:00.0000000-07:00' --> 3/30/2013 6:00:00 PM Local

```

However, if you restore the data on a system in a different time zone, only the date and time value that was formatted with the "o" (round-trip) standard format string preserves time zone information and therefore represents the same instant in time. Here's the output when the date and time data is restored on a system in the Romance Standard Time zone:

Console

```
'3/30/2023 6:00:00 PM' --> 3/30/2023 6:00:00 PM Unspecified
'2023-03-30T18:00:00' --> 3/30/2023 6:00:00 PM Unspecified
'2023-03-30T18:00:00.0000000-07:00' --> 3/31/2023 3:00:00 AM Local
```

To accurately reflect a date and time value that represents a single moment of time regardless of the time zone of the system on which the data is deserialized, you can do any of the following:

- Save the value as a string by using the "o" (round-trip) standard format string. Then deserialize it on the target system.
- Convert it to UTC and save it as a string by using the "r" (RFC1123) standard format string. Then deserialize it on the target system and convert it to local time.
- Convert it to UTC and save it as a string by using the "u" (universal sortable) standard format string. Then deserialize it on the target system and convert it to local time.

The following example illustrates each technique.

C#

```
using System;
using System.IO;

public class Example9
{
    public static void Main9()
    {
        // Serialize a date.
        DateTime dateOriginal = new DateTime(2023, 3, 30, 18, 0, 0);
        dateOriginal = DateTime.SpecifyKind(dateOriginal,
DateTimeKind.Local);

        // Serialize the date in string form.
        if (!File.Exists("DateInfo2.dat"))
        {
            StreamWriter sw = new StreamWriter("DateInfo2.dat");
            sw.WriteLine("{0:o}|{1:r}|{1:u}", dateOriginal,
                        dateOriginal.ToUniversalTime());
            sw.Close();
        }

        // Restore the date from string values.
        StreamReader sr = new StreamReader("DateInfo2.dat");
        string datesToSplit = sr.ReadToEnd();
        string[] dateStrings = datesToSplit.Split('|');
        for (int ctr = 0; ctr < dateStrings.Length; ctr++)
        {
```

```
DateTime newDate = DateTime.Parse(dateStrings[ctr]);
if (ctr == 1)
{
    Console.WriteLine($"'{dateStrings[ctr]}' --> {newDate}
{newDate.Kind}");
}
else
{
    DateTime newLocalDate = newDate.ToLocalTime();
    Console.WriteLine($"'{dateStrings[ctr]}' --> {newLocalDate}
{newLocalDate.Kind}");
}
}
```

When the data is serialized on a system in the Pacific Standard Time zone and deserialized on a system in the Romance Standard Time zone, the example displays the following output:

Console

```
'2023-03-30T18:00:00.0000000-07:00' --> 3/31/2023 3:00:00 AM Local
'Sun, 31 Mar 2023 01:00:00 GMT' --> 3/31/2023 3:00:00 AM Local
'2023-03-31 01:00:00Z' --> 3/31/2023 3:00:00 AM Local
```

For more information, see [Convert times between time zones](#).

Perform date and time arithmetic

Both the [DateTime](#) and [DateTimeOffset](#) types support arithmetic operations. You can calculate the difference between two date values, or you can add or subtract particular time intervals to or from a date value. However, arithmetic operations on date and time values do not take time zones and time zone adjustment rules into account. Because of this, date and time arithmetic on values that represent moments in time can return inaccurate results.

For example, the transition from Pacific Standard Time to Pacific Daylight Time occurs on the second Sunday of March, which is March 10 for the year 2013. As the following example shows, if you calculate the date and time that is 48 hours after March 9, 2013 at 10:30 A.M. on a system in the Pacific Standard Time zone, the result, March 11, 2013 at 10:30 A.M., does not take the intervening time adjustment into account.

C#

```

using System;

public class Example7
{
    public static void Main7()
    {
        DateTime date1 = DateTime.SpecifyKind(new DateTime(2013, 3, 9, 10,
30, 0),
                                            DateTimeKind.Local);
        TimeSpan interval = new TimeSpan(48, 0, 0);
        DateTime date2 = date1 + interval;
        Console.WriteLine("{0:g} + {1:N1} hours = {2:g}",
                          date1, interval.TotalHours, date2);
    }
}

// The example displays the following output:
//      3/9/2013 10:30 AM + 48.0 hours = 3/11/2013 10:30 AM

```

To ensure that an arithmetic operation on date and time values produces accurate results, follow these steps:

1. Convert the time in the source time zone to UTC.
2. Perform the arithmetic operation.
3. If the result is a date and time value, convert it from UTC to the time in the source time zone.

The following example is similar to the previous example, except that it follows these three steps to correctly add 48 hours to March 9, 2013 at 10:30 A.M.

C#

```

using System;

public class Example8
{
    public static void Main8()
    {
        TimeZoneInfo pst = TimeZoneInfo.FindSystemTimeZoneById("Pacific
Standard Time");
        DateTime date1 = DateTime.SpecifyKind(new DateTime(2013, 3, 9, 10,
30, 0),
                                            DateTimeKind.Local);
        DateTime utc1 = date1.ToUniversalTime();
        TimeSpan interval = new TimeSpan(48, 0, 0);
        DateTime utc2 = utc1 + interval;
        DateTime date2 = TimeZoneInfo.ConvertTimeFromUtc(utc2, pst);
        Console.WriteLine("{0:g} + {1:N1} hours = {2:g}",
                          date1, interval.TotalHours, date2);
    }
}

```

```
}
```

```
// The example displays the following output:
//      3/9/2013 10:30 AM + 48.0 hours = 3/11/2013 11:30 AM
```

For more information, see [Perform arithmetic operations with dates and times](#).

Use culture-sensitive names for date elements

Your app may need to display the name of the month or the day of the week. To do this, code such as the following is common.

```
C#
```

```
using System;

public class Example12
{
    public static void Main12()
    {
        DateTime midYear = new DateTime(2013, 7, 1);
        Console.WriteLine("{0:d} is a {1}.", midYear, GetDayName(midYear));
    }

    private static string GetDayName(DateTime date)
    {
        return date.DayOfWeek.ToString("G");
    }
}

// The example displays the following output:
//      7/1/2013 is a Monday.
```

However, this code always returns the names of the days of the week in English. Code that extracts the name of the month is often even more inflexible. It frequently assumes a twelve-month calendar with names of months in a specific language.

By using [custom date and time format strings](#) or the properties of the [DateTimeFormatInfo](#) object, it is easy to extract strings that reflect the names of days of the week or months in the user's culture, as the following example illustrates. It changes the current culture to French (France) and displays the name of the day of the week and the name of the month for July 1, 2013.

```
C#
```

```
using System;
using System.Globalization;
```

```

public class Example13
{
    public static void Main13()
    {
        // Set the current culture to French (France).
        CultureInfo.CurrentCulture = CultureInfo.CreateSpecificCulture("fr-
FR");

        DateTime midYear = new DateTime(2013, 7, 1);
        Console.WriteLine("{0:d} is a {1}.", midYear,
DateUtilities.GetDayName(midYear));
        Console.WriteLine("{0:d} is a {1}.", midYear,
DateUtilities.GetDayName((int)midYear.DayOfWeek));
        Console.WriteLine("{0:d} is in {1}.", midYear,
DateUtilities.GetMonthName(midYear));
        Console.WriteLine("{0:d} is in {1}.", midYear,
DateUtilities.GetMonthName(midYear.Month));
    }
}

public static class DateUtilities
{
    public static string GetDayName(int dayOfWeek)
    {
        if (dayOfWeek < 0 | dayOfWeek >
DateTimeFormatInfo.CurrentInfo.DayNames.Length)
            return String.Empty;
        else
            return DateTimeFormatInfo.CurrentInfo.DayNames[dayOfWeek];
    }

    public static string GetDayName(DateTime date)
    {
        return date.ToString("dddd");
    }

    public static string GetMonthName(int month)
    {
        if (month < 1 | month >
DateTimeFormatInfo.CurrentInfo.MonthNames.Length - 1)
            return String.Empty;
        else
            return DateTimeFormatInfo.CurrentInfo.MonthNames[month - 1];
    }

    public static string GetMonthName(DateTime date)
    {
        return date.ToString("MMMM");
    }
}

// The example displays the following output:
//      01/07/2013 is a lundi.
//      01/07/2013 is a lundi.

```

```
//      01/07/2013 is in juillet.  
//      01/07/2013 is in juillet.
```

Numeric values

The handling of numbers depends on whether they are displayed in the user interface or persisted. This section examines both usages.

① Note

In parsing and formatting operations, .NET recognizes only the Basic Latin characters 0 through 9 (U+0030 through U+0039) as numeric digits.

Display numeric values

Typically, when numbers are displayed in the user interface, you should use the formatting conventions of the user's culture, which is defined by the [CultureInfo.CurrentCulture](#) property and by the [NumberFormatInfo](#) object returned by the [CultureInfo.CurrentCulture.NumberFormat](#) property. The formatting conventions of the current culture are automatically used when you format a date in the following ways:

- Using the parameterless `ToString` method of any numeric type.
- Using the `ToString(String)` method of any numeric type, which includes a format string as an argument.
- Using [composite formatting](#) with numeric values.

The following example displays the average temperature per month in Paris, France. It first sets the current culture to French (France) before displaying the data, and then sets it to English (United States). In each case, the month names and temperatures are displayed in the format that is appropriate for that culture. Note that the two cultures use different decimal separators in the temperature value. Also note that the example uses the "MMMM" custom date and time format string to display the full month name, and that it allocates the appropriate amount of space for the month name in the result string by determining the length of the longest month name in the [DateTimeFormatInfo.MonthNames](#) array.

C#

```
using System;  
using System.Globalization;  
using System.Threading;
```

```

public class Example14
{
    public static void Main14()
    {
        DateTime dateForMonth = new DateTime(2013, 1, 1);
        double[] temperatures = { 3.4, 3.5, 7.6, 10.4, 14.5, 17.2,
                                  19.9, 18.2, 15.9, 11.3, 6.9, 5.3 };

        Thread.CurrentThread.CurrentCulture =
CultureInfo.CreateSpecificCulture("fr-FR");
        Console.WriteLine("Current Culture: {0}",
CultureInfo.CurrentCulture.DisplayName);
        // Build the format string dynamically so we allocate enough space
for the month name.
        string fmtString = "{0,-" + GetLongestMonthNameLength().ToString() +
":MMMM}" {1,4}";
        for (int ctr = 0; ctr < temperatures.Length; ctr++)
            Console.WriteLine(fmtString,
dateForMonth.AddMonths(ctr),
temperatures[ctr]);

        Console.WriteLine();

        Thread.CurrentThread.CurrentCulture =
CultureInfo.CreateSpecificCulture("en-US");
        Console.WriteLine("Current Culture: {0}",
CultureInfo.CurrentCulture.DisplayName);
        fmtString = "{0,-" + GetLongestMonthNameLength().ToString() +
":MMMM}" {1,4}";
        for (int ctr = 0; ctr < temperatures.Length; ctr++)
            Console.WriteLine(fmtString,
dateForMonth.AddMonths(ctr),
temperatures[ctr]);
    }

    private static int GetLongestMonthNameLength()
    {
        int length = 0;
        foreach (var nameOfMonth in
DateTimeFormatInfo.CurrentInfo.MonthNames)
            if (nameOfMonth.Length > length) length = nameOfMonth.Length;

        return length;
    }
}

// The example displays the following output:
// Current Culture: French (France)
//      janvier      3,4
//      février      3,5
//      mars         7,6
//      avril         10,4
//      mai          14,5
//      juin          17,2

```

```
//      juillet      19,9
//      août         18,2
//      septembre   15,9
//      octobre      11,3
//      novembre     6,9
//      décembre     5,3
//
//      Current Culture: English (United States)
//      January       3.4
//      February      3.5
//      March         7.6
//      April         10.4
//      May           14.5
//      June          17.2
//      July          19.9
//      August        18.2
//      September     15.9
//      October       11.3
//      November      6.9
//      December       5.3
```

Persist numeric values

You should never persist numeric data in a culture-specific format. This is a common programming error that results in either corrupted data or a run-time exception. The following example generates ten random floating-point numbers, and then serializes them as strings by using the formatting conventions of the English (United States) culture. When the data is retrieved and parsed by using the conventions of the English (United States) culture, it is successfully restored. However, when it is retrieved and parsed by using the conventions of the French (France) culture, none of the numbers can be parsed because the cultures use different decimal separators.

C#

```
using System;
using System.Globalization;
using System.IO;
using System.Threading;

public class Example15
{
    public static void Main15()
    {
        // Create ten random doubles.
        Thread.CurrentThread.CurrentCulture =
CultureInfo.CreateSpecificCulture("en-US");
        double[] numbers = GetRandomNumbers(10);
        DisplayRandomNumbers(numbers);
```

```

        // Persist the numbers as strings.
        StreamWriter sw = new StreamWriter("randoms.dat");
        for (int ctr = 0; ctr < numbers.Length; ctr++)
            sw.Write("{0:R}{1}", numbers[ctr], ctr < numbers.Length - 1 ?
"|" : "");
    }

    sw.Close();

    // Read the persisted data.
    StreamReader sr = new StreamReader("randoms.dat");
    string numericData = sr.ReadToEnd();
    sr.Close();
    string[] numberStrings = numericData.Split('|');

    // Restore and display the data using the conventions of the en-US
    culture.
    Console.WriteLine("Current Culture: {0}",
                      Thread.CurrentThread.CurrentCulture.DisplayName);
    foreach (var numberStr in numberStrings)
    {
        double restoredNumber;
        if (Double.TryParse(numberStr, out restoredNumber))
            Console.WriteLine(restoredNumber.ToString("R"));
        else
            Console.WriteLine("ERROR: Unable to parse '{0}'",
numberStr);
    }
    Console.WriteLine();

    // Restore and display the data using the conventions of the fr-FR
    culture.
    Thread.CurrentThread.CurrentCulture =
CultureInfo.CreateSpecificCulture("fr-FR");
    Console.WriteLine("Current Culture: {0}",
                      Thread.CurrentThread.CurrentCulture.DisplayName);
    foreach (var numberStr in numberStrings)
    {
        double restoredNumber;
        if (Double.TryParse(numberStr, out restoredNumber))
            Console.WriteLine(restoredNumber.ToString("R"));
        else
            Console.WriteLine("ERROR: Unable to parse '{0}'",
numberStr);
    }
}

private static double[] GetRandomNumbers(int n)
{
    Random rnd = new Random();
    double[] numbers = new double[n];
    for (int ctr = 0; ctr < n; ctr++)
        numbers[ctr] = rnd.NextDouble() * 1000;
    return numbers;
}

```

```

private static void DisplayRandomNumbers(double[] numbers)
{
    for (int ctr = 0; ctr < numbers.Length; ctr++)
        Console.WriteLine(numbers[ctr].ToString("R"));
    Console.WriteLine();
}
}

// The example displays output like the following:
//      487.0313743534644
//      674.12000879371533
//      498.72077885024288
//      42.3034229512808
//      970.57311049223563
//      531.33717716268131
//      587.82905693530529
//      562.25210175023039
//      600.7711019370571
//      299.46113717717174
//
//      Current Culture: English (United States)
//      487.0313743534644
//      674.12000879371533
//      498.72077885024288
//      42.3034229512808
//      970.57311049223563
//      531.33717716268131
//      587.82905693530529
//      562.25210175023039
//      600.7711019370571
//      299.46113717717174
//
//      Current Culture: French (France)
//      ERROR: Unable to parse '487.0313743534644'
//      ERROR: Unable to parse '674.12000879371533'
//      ERROR: Unable to parse '498.72077885024288'
//      ERROR: Unable to parse '42.3034229512808'
//      ERROR: Unable to parse '970.57311049223563'
//      ERROR: Unable to parse '531.33717716268131'
//      ERROR: Unable to parse '587.82905693530529'
//      ERROR: Unable to parse '562.25210175023039'
//      ERROR: Unable to parse '600.7711019370571'
//      ERROR: Unable to parse '299.46113717717174'

```

To avoid this problem, you can use one of these techniques:

- Save and parse the string representation of the number by using a custom format string that is the same regardless of the user's culture.
- Save the number as a string by using the formatting conventions of the invariant culture, which is returned by the [CultureInfo.InvariantCulture](#) property.

Serializing currency values is a special case. Because a currency value depends on the unit of currency in which it's expressed, it makes little sense to treat it as an independent numeric value. However, if you save a currency value as a formatted string that includes a currency symbol, it cannot be deserialized on a system whose default culture uses a different currency symbol, as the following example shows.

C#

```
using System;
using System.Globalization;
using System.IO;
using System.Threading;

public class Example1
{
    public static void Main()
    {
        // Display the currency value.
        Thread.CurrentThread.CurrentCulture =
CultureInfo.CreateSpecificCulture("en-US");
        Decimal value = 16039.47m;
        Console.WriteLine("Current Culture: {0}",
CultureInfo.CurrentCulture.DisplayName);
        Console.WriteLine("Currency Value: {0:C2}", value);

        // Persist the currency value as a string.
        StreamWriter sw = new StreamWriter("currency.dat");
        sw.Write(value.ToString("C2"));
        sw.Close();

        // Read the persisted data using the current culture.
        StreamReader sr = new StreamReader("currency.dat");
        string currencyData = sr.ReadToEnd();
        sr.Close();

        // Restore and display the data using the conventions of the current
culture.
        Decimal restoredValue;
        if (Decimal.TryParse(currencyData, out restoredValue))
            Console.WriteLine(restoredValue.ToString("C2"));
        else
            Console.WriteLine("ERROR: Unable to parse '{0}'", currencyData);
        Console.WriteLine();

        // Restore and display the data using the conventions of the en-GB
culture.
        Thread.CurrentThread.CurrentCulture =
CultureInfo.CreateSpecificCulture("en-GB");
        Console.WriteLine("Current Culture: {0}",
                Thread.CurrentThread.CurrentCulture.DisplayName);
        if (Decimal.TryParse(currencyData, NumberStyles.Currency, null, out
restoredValue))
            Console.WriteLine(restoredValue.ToString("C2"));
```

```

        else
            Console.WriteLine("ERROR: Unable to parse '{0}'", currencyData);
        Console.WriteLine();
    }
}

// The example displays output like the following:
//      Current Culture: English (United States)
//      Currency Value: $16,039.47
//      ERROR: Unable to parse '$16,039.47'
//
//      Current Culture: English (United Kingdom)
//      ERROR: Unable to parse '$16,039.47'

```

Instead, you should serialize the numeric value along with some cultural information, such as the name of the culture, so that the value and its currency symbol can be deserialized independently of the current culture. The following example does that by defining a `CurrencyValue` structure with two members: the `Decimal` value and the name of the culture to which the value belongs.

C#

```

using System;
using System.Globalization;
using System.Text.Json;
using System.Threading;

public class Example2
{
    public static void Main2()
    {
        // Display the currency value.
        Thread.CurrentThread.CurrentCulture =
CultureInfo.CreateSpecificCulture("en-US");
        Decimal value = 16039.47m;
        Console.WriteLine($"Current Culture:
{CultureInfo.CurrentCulture.DisplayName}");
        Console.WriteLine($"Currency Value: {value:C2}");

        // Serialize the currency data.
        CurrencyValue data = new()
        {
            Amount = value,
            CultureName = CultureInfo.CurrentCulture.Name
        };
        string serialized = JsonSerializer.Serialize(data);
        Console.WriteLine();

        // Change the current culture.
        CultureInfo.CurrentCulture = CultureInfo.CreateSpecificCulture("en-
GB");
        Console.WriteLine($"Current Culture:
{CultureInfo.CurrentCulture.DisplayName}");

```

```

    // Deserialize the data.
    CurrencyValue restoredData =
JsonSerializer.Deserialize<CurrencyValue>(serialized);

    // Display the round-tripped value.
    CultureInfo culture =
CultureInfo.CreateSpecificCulture(restoredData.CultureName);
    Console.WriteLine($"Currency Value:
{restoredData.Amount.ToString("C2", culture)}");
}
}

internal struct CurrencyValue
{
    public decimal Amount { get; set; }
    public string CultureName { get; set; }
}

// The example displays the following output:
//      Current Culture: English (United States)
//      Currency Value: $16,039.47
//
//      Current Culture: English (United Kingdom)
//      Currency Value: $16,039.47

```

Work with culture-specific settings

In .NET, the [CultureInfo](#) class represents a particular culture or region. Some of its properties return objects that provide specific information about some aspect of a culture:

- The [CultureInfo.CompareInfo](#) property returns a [CompareInfo](#) object that contains information about how the culture compares and orders strings.
- The [CultureInfo.DateTimeFormat](#) property returns a [DateTimeFormatInfo](#) object that provides culture-specific information used in formatting date and time data.
- The [CultureInfo.NumberFormat](#) property returns a [NumberFormatInfo](#) object that provides culture-specific information used in formatting numeric data.
- The [CultureInfo.TextInfo](#) property returns a [TextInfo](#) object that provides information about the culture's writing system.

In general, do not make any assumptions about the values of specific [CultureInfo](#) properties and their related objects. Instead, you should view culture-specific data as subject to change, for these reasons:

- Individual property values are subject to change and revision over time, as data is corrected, better data becomes available, or culture-specific conventions change.
- Individual property values may vary across versions of .NET or operating system versions.
- .NET supports replacement cultures. This makes it possible to define a new custom culture that either supplements existing standard cultures or completely replaces an existing standard culture.
- On Windows systems, the user can customize culture-specific settings by using the **Region and Language** app in Control Panel. When you instantiate a [CultureInfo](#) object, you can determine whether it reflects these user customizations by calling the [CultureInfo\(String, Boolean\)](#) constructor. Typically, for end-user apps, you should respect user preferences so that the user is presented with data in a format that they expect.

See also

- [Globalization and localization](#)
- [Best practices for using strings](#)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

.NET globalization and ICU

Article • 08/29/2023

Before .NET 5, the .NET globalization APIs used different underlying libraries on different platforms. On Unix, the APIs used [International Components for Unicode \(ICU\)](#), and on Windows, they used [National Language Support \(NLS\)](#). This resulted in some behavioral differences in a handful of globalization APIs when running applications on different platforms. Behavior differences were evident in these areas:

- Cultures and culture data
- String casing
- String sorting and searching
- Sort keys
- String normalization
- Internationalized Domain Names (IDN) support
- Time zone display name on Linux

Starting with .NET 5, developers have more control over which underlying library is used, enabling applications to avoid differences across platforms.

ICU on Windows

Windows now incorporates a preinstalled [icu.dll](#) version as part of its features that's automatically employed for globalization tasks. This modification allows .NET to leverage this ICU library for its globalization support. In cases where the ICU library is unavailable or cannot be loaded, as is the case with older Windows versions, .NET 5 and subsequent versions revert to using the NLS-based implementation.

The following table shows which versions of .NET are capable of loading the ICU library across different Windows client and server versions:

[] Expand table

| .NET version | Windows version |
|------------------|---|
| .NET 5 or .NET 6 | Windows client 10 version 1903 or later |
| .NET 5 or .NET 6 | Windows Server 2022 or later |
| .NET 7 or later | Windows client 10 version 1703 or later |
| .NET 7 or later | Windows Server 2019 or later |

ⓘ Note

.NET 7 and later versions have the capability to load ICU on older Windows versions, in contrast to .NET 6 and .NET 5.

ⓘ Note

Even when using ICU, the `CurrentCulture`, `CurrentUICulture`, and `CurrentRegion` members still use Windows operating system APIs to honor user settings.

Behavioral differences

If you upgrade your app to target .NET 5 or later, you might see changes in your app even if you don't realize you're using globalization facilities. This section lists one of the behavioral changes you might see, but there are others too.

String.IndexOf

Consider the following code that calls `String.IndexOf(String)` to find the index of the null character `\0` in a string.

C#

```
const string greeting = "He\0llo";
Console.WriteLine(${greeting.IndexOf("\0")});
Console.WriteLine(${greeting.IndexOf("\0",
 StringComparison.CurrentCulture)});
Console.WriteLine(${greeting.IndexOf("\0", StringComparison.Ordinal)});
```

- In .NET Core 3.1 and earlier versions on Windows, the snippet prints `3` on each of the three lines.
- For .NET 5 and subsequent versions running on the Windows versions listed in the [ICU on Windows](#) section table, the snippet prints `0`, `0`, and `3` (for the ordinal search).

By default, `String.IndexOf(String)` performs a culture-aware linguistic search. ICU considers the null character `\0` to be a *zero-weight character*, and thus the character isn't found in the string when using a linguistic search on .NET 5 and later. However, NLS doesn't consider the null character `\0` to be a zero-weight character, and a linguistic

search on .NET Core 3.1 and earlier locates the character at position 3. An ordinal search finds the character at position 3 on all .NET versions.

You can run code analysis rules [CA1307: Specify StringComparison for clarity](#) and [CA1309: Use ordinal StringComparison](#) to find call sites in your code where the string comparison isn't specified or it is not ordinal.

For more information, see [Behavior changes when comparing strings on .NET 5+](#).

String.EndsWith

C#

```
const string foo = "abc";

Console.WriteLine(foo.EndsWith("\0"));
Console.WriteLine(foo.EndsWith("c"));
Console.WriteLine(foo.EndsWith("\0", StringComparison.CurrentCulture));
Console.WriteLine(foo.EndsWith("\0", StringComparison.Ordinal));
Console.WriteLine(foo.EndsWith('\0'));
```

ⓘ Important

In .NET 5+ running on Windows versions listed in the [ICU on Windows](#) table, the preceding snippet prints:

Output

```
True
True
True
False
False
```

To avoid this behavior, use the `char` parameter overload or `StringComparison.Ordinal`.

String.StartsWith

C#

```
const string foo = "abc";

Console.WriteLine(foo.StartsWith("\0"));
Console.WriteLine(foo.StartsWith("a"));
```

```
Console.WriteLine(foo.StartsWith("\0", StringComparison.CurrentCulture));
Console.WriteLine(foo.StartsWith("\0", StringComparison.Ordinal));
Console.WriteLine(foo.StartsWith('\0'));
```

ⓘ Important

In .NET 5+ running on Windows versions listed in the [ICU on Windows](#) table, the preceding snippet prints:

Output

```
True
True
True
False
False
```

To avoid this behavior, use the `char` parameter overload or `StringComparison.Ordinal`.

TimeZoneInfo.FindSystemTimeZoneById

ICU provides the flexibility to create `TimeZoneInfo` instances using [IANA](#) time zone IDs, even when the application is running on Windows. Similarly, you can create `TimeZoneInfo` instances with Windows time zone IDs, even when running on non-Windows platforms. However, it's important to note that this functionality isn't available when using [NLS mode](#) or [globalization invariant mode](#).

ICU dependent APIs

.NET introduced APIs that are dependent on ICU. These APIs can succeed only when using ICU. Here are some examples:

- [TryConvertIanaIdToWindowsId\(String, String\)](#)
- [TryConvertWindowsIdToIanaId](#)

On the Windows versions listed in the [ICU on Windows](#) section table, the mentioned APIs will consistently succeed. However, on older versions of Windows, these APIs will consistently fail. In such cases, you can enable the [app-local ICU](#) feature to ensure the success of these APIs. On non-Windows platforms, these APIs will always succeed regardless of the version.

In addition, it's crucial for apps to ensure that they're not running in [globalization invariant mode](#) or [NLS mode](#) to guarantee the success of these APIs.

Use NLS instead of ICU

Using ICU instead of NLS may result in behavioral differences with some globalization-related operations. To revert back to using NLS, a developer can opt out of the ICU implementation. Applications can enable NLS mode in any of the following ways:

- In the project file:

XML

```
<ItemGroup>
  <RuntimeHostConfigurationOption Include="System.Globalization.UseNls"
    Value="true" />
</ItemGroup>
```

- In the `runtimeconfig.json` file:

JSON

```
{
  "runtimeOptions": {
    "configProperties": {
      "System.Globalization.UseNls": true
    }
  }
}
```

- By setting the environment variable `DOTNET_SYSTEM_GLOBALIZATION_USENLS` to the value `true` or `1`.

ⓘ Note

A value set in the project or in the `runtimeconfig.json` file takes precedence over the environment variable.

For more information, see [Runtime config settings](#).

Determine if your app is using ICU

The following code snippet can help you determine if your app is running with ICU libraries (and not NLS).

C#

```
public static bool ICUMode()
{
    SortVersion sortVersion =
CultureInfo.InvariantCulture.CompareInfo.Version;
    byte[] bytes = sortVersion.SortId.ToByteArray();
    int version = bytes[3] << 24 | bytes[2] << 16 | bytes[1] << 8 |
bytes[0];
    return version != 0 && version == sortVersion.FullVersion;
}
```

To determine the version of .NET, use [RuntimeInformation.FrameworkDescription](#).

App-local ICU

Each release of ICU may bring with it bug fixes as well as updated Common Locale Data Repository (CLDR) data that describes the world's languages. Moving between versions of ICU can subtly impact app behavior when it comes to globalization-related operations. To help application developers ensure consistency across all deployments, .NET 5 and later versions enable apps on both Windows and Unix to carry and use their own copy of ICU.

Applications can opt-in to an app-local ICU implementation mode in one of the following ways:

- In the project file, set the appropriate `RuntimeHostConfigurationOption` value:

XML

```
<ItemGroup>
    <RuntimeHostConfigurationOption
        Include="System.Globalization.AppLocalIcu" Value="<suffix>:<version> or
<version>" />
</ItemGroup>
```

- Or in the `runtimeconfig.json` file, set the appropriate

`runtimeOptions.configProperties` value:

JSON

```
{
    "runtimeOptions": {
```

```
        "configProperties": {
            "System.Globalization.AppLocalIcu": "<suffix>:<version> or
<version>"
        }
    }
}
```

- Or by setting the environment variable `DOTNET_SYSTEM_GLOBALIZATION_APPLOCALICU` to the value `<suffix>:<version>` or `<version>`.

`<suffix>`: Optional suffix of fewer than 36 characters in length, following the public ICU packaging conventions. When building a custom ICU, you can customize it to produce the lib names and exported symbol names to contain a suffix, for example, `libicuucmyapp`, where `myapp` is the suffix.

`<version>`: A valid ICU version, for example, 67.1. This version is used to load the binaries and to get the exported symbols.

When either of these options is set, you can add a [Microsoft.ICU.ICU4C.Runtime](#)  `PackageReference` to your project that corresponds to the configured `version` and that's all that is needed.

Alternatively, to load ICU when the app-local switch is set, .NET uses the [NativeLibrary.TryLoad](#) method, which probes multiple paths. The method first tries to find the library in the `NATIVE_DLL_SEARCH_DIRECTORIES` property, which is created by the dotnet host based on the `deps.json` file for the app. For more information, see [Default probing](#).

For self-contained apps, no special action is required by the user, other than making sure ICU is in the app directory (for self-contained apps, the working directory defaults to `NATIVE_DLL_SEARCH_DIRECTORIES`).

If you're consuming ICU via a NuGet package, this works in framework-dependent applications. NuGet resolves the native assets and includes them in the `deps.json` file and in the output directory for the application under the `runtimes` directory. .NET loads it from there.

For framework-dependent apps (not self-contained) where ICU is consumed from a local build, you must take additional steps. The .NET SDK doesn't yet have a feature for "loose" native binaries to be incorporated into `deps.json` (see [this SDK issue](#) ). Instead, you can enable this by adding additional information into the application's project file. For example:

XML

```

<ItemGroup>
  <IcuAssemblies Include="icu\*.so*" />
  <RuntimeTargetsCopyLocalItems Include="@{IcuAssemblies}" AssetType="native" CopyLocal="true"
    DestinationSubDirectory="runtimes/linux-x64/native/"
    DestinationSubPath"%(FileName)%(Extension)"
    RuntimeIdentifier="linux-x64"
    NuGetPackageId="System.Private.Runtime.UnicodeData" />
</ItemGroup>

```

This must be done for all the ICU binaries for the supported runtimes. Also, the `NuGetPackageId` metadata in the `RuntimeTargetsCopyLocalItems` item group needs to match a NuGet package that the project actually references.

macOS behavior

macOS has a different behavior for resolving dependent dynamic libraries from the load commands specified in the `Mach-O` file than the Linux loader. In the Linux loader, .NET can try `libcudata`, `libicuuc`, and `libicui18n` (in that order) to satisfy ICU dependency graph. However, on macOS, this doesn't work. When building ICU on macOS, you, by default, get a dynamic library with these load commands in `libicuuc`. The following snippet shows an example.

```

sh

~/ % otool -L /Users/santifdezr/repos/icu-build/icu/install/lib/libicuuc.67.1.dylib
/Users/santifdezr/repos/icu-build/icu/install/lib/libicuuc.67.1.dylib:
  libicuuc.67.dylib (compatibility version 67.0.0, current version 67.1.0)
  libicudata.67.dylib (compatibility version 67.0.0, current version 67.1.0)
  /usr/lib/libSystem.B.dylib (compatibility version 1.0.0, current version
  1281.100.1)
  /usr/lib/libc++.1.dylib (compatibility version 1.0.0, current version
  902.1.0)

```

These commands just reference the name of the dependent libraries for the other components of ICU. The loader performs the search following the `dlopen` conventions, which involves having these libraries in the system directories or setting the `LD_LIBRARY_PATH` env vars, or having ICU at the app-level directory. If you can't set `LD_LIBRARY_PATH` or ensure that ICU binaries are at the app-level directory, you will need to do some extra work.

There are some directives for the loader, like `@loader_path`, which tell the loader to search for that dependency in the same directory as the binary with that load command.

There are two ways to achieve this:

- `install_name_tool -change`

Run the following commands:

Bash

```
install_name_tool -change "libcudata.67.dylib"
"@loader_path/libicudata.67.dylib" /path/to/libicuuc.67.1.dylib
install_name_tool -change "libcudata.67.dylib"
"@loader_path/libicudata.67.dylib" /path/to/libicui18n.67.1.dylib
install_name_tool -change "libicuuc.67.dylib"
"@loader_path/libicuuc.67.dylib" /path/to/libicui18n.67.1.dylib
```

- Patch ICU to produce the install names with `@loader_path`

Before running autoconf (`./runConfigureICU`), change [these lines ↗](#) to:

```
LD SONAME = -Wl,-compatibility_version -Wl,$(SO_TARGET_VERSION_MAJOR) -
Wl,-current_version -Wl,$(SO_TARGET_VERSION) -install_name
@loader_path/${notdir $(MIDDLE_SO_TARGET)}
```

ICU on WebAssembly

A version of ICU is available that's specifically for WebAssembly workloads. This version provides globalization compatibility with desktop profiles. To reduce the ICU data file size from 24 MB to 1.4 MB (or ~0.3 MB if compressed with Brotli), this workload has a handful of limitations.

The following APIs are not supported:

- [CultureInfo.EnglishName](#)
- [CultureInfo.NativeName](#)
- [DateTimeFormatInfo.NativeCalendarName](#)
- [RegionInfo.NativeName](#)

The following APIs are supported with limitations:

- [String.Normalize\(NormalizationForm\)](#) and [String.IsNormalized\(NormalizationForm\)](#) don't support the rarely used [FormKC](#) and [FormKD](#) forms.
- [RegionInfo.CurrencyNativeName](#) returns the same value as [RegionInfo.CurrencyEnglishName](#).

In addition, fewer locales are supported. The supported list can be found in the [dotnet/icu repo ↗](#).

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

.NET is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

Perform culture-insensitive string operations

Article • 03/18/2023

Culture-sensitive string operations are advantageous if you're creating applications designed to display results to users on a per-culture basis. By default, culture-sensitive methods obtain the culture to use from the [CurrentCulture](#) property for the current thread.

Sometimes, culture-sensitive string operations are not the desired behavior. Using culture-sensitive operations when results should be independent of culture can cause application code to fail on cultures with custom case mappings and sorting rules. For an example, see the [String Comparisons that Use the Current Culture](#) section in [Best Practices for Using Strings](#).

Whether string operations should be culture-sensitive or culture-insensitive depends on how your application uses the results. String operations that display results to the user should typically be culture-sensitive. For example, if an application displays a sorted list of localized strings in a list box, the application should perform a culture-sensitive sort.

Results of string operations that are used internally should typically be culture-insensitive. In general, if the application is working with file names, persistence formats, or symbolic information that is not displayed to the user, results of string operations should not vary by culture. For example, if an application compares a string to determine whether it is a recognized XML tag, the comparison should not be culture-sensitive. In addition, if a security decision is based on the result of a string comparison or case change operation, the operation should be culture-insensitive to ensure that the result is not affected by the value of [CurrentCulture](#).

Most .NET methods that *by default* perform culture-sensitive string operations also provide an overload that allows you to guarantee culture-insensitive results. These overloads that take a [CultureInfo](#) argument allow you to eliminate cultural variations in case mappings and sorting rules. For culture-insensitive string operations, specify the culture as [CultureInfo.InvariantCulture](#).

In this section

The articles in this section demonstrate how to perform culture-insensitive string operations using .NET methods that are culture-sensitive by default.

Performing culture-insensitive string comparisons

Describes how to use the [String.Compare](#) and [String.CompareTo](#) methods to perform culture-insensitive string comparisons.

Performing culture-insensitive case changes

Describes how to use the [String.ToUpper](#), [String.ToLower](#), [Char.ToUpper](#), and [Char.ToLower](#) methods to perform culture-insensitive case changes.

Performing culture-insensitive string operations in collections

Describes how to use the [CaseInsensitiveComparer](#), [CaseInsensitiveHashCodeProvider](#) class, [SortedList](#), [ArrayList.Sort](#) and [CollectionsUtil.CreateCaseInsensitiveHashtable](#) to perform culture-insensitive operations in collections.

Performing culture-insensitive string operations in arrays

Describes how to use the [Array.Sort](#) and [Array.BinarySearch](#) methods to perform culture-insensitive operations in arrays.

See also

- [Sorting Weight Tables \(for .NET on Windows systems\)](#) ↗
- [Default Unicode Collation Element Table \(for .NET Core on Linux and macOS\)](#) ↗

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

Perform culture-insensitive string comparisons

Article • 03/18/2023

By default, the [String.Compare](#) method performs culture-sensitive and case-sensitive comparisons. This method also includes several overloads that provide a `culture` parameter that lets you specify the culture to use, and a `comparisonType` parameter that lets you specify the comparison rules to use. Calling these methods instead of the default overload removes any ambiguity about the rules used in a particular method call, and makes it clear whether a particular comparison is culture-sensitive or culture-insensitive.

ⓘ Note

Both overloads of the [String.CompareTo](#) method perform culture-sensitive and case-sensitive comparisons; you cannot use this method to perform culture-insensitive comparisons. For code clarity, we recommend that you use the [String.Compare](#) method instead.

For culture-sensitive operations, specify the [StringComparison.CurrentCulture](#) or [StringComparison.CurrentCultureIgnoreCase](#) enumeration value as the `comparisonType` parameter. If you want to perform a culture-sensitive comparison using a designated culture other than the current culture, specify the [CultureInfo](#) object that represents that culture as the `culture` parameter.

The culture-insensitive string comparisons supported by the [String.Compare](#) method are either linguistic (based on the sorting conventions of the invariant culture) or non-linguistic (based on the ordinal value of the characters in the string). Most culture-insensitive string comparisons are non-linguistic. For these comparisons, specify the [StringComparison.Ordinal](#) or [StringComparison.OrdinalIgnoreCase](#) enumeration value as the `comparisonType` parameter. For example, if a security decision (such as a user name or password comparison) is based on the result of a string comparison, the operation should be culture-insensitive and non-linguistic to ensure that the result is not affected by the conventions of a particular culture or language.

Use culture-insensitive linguistic string comparison if you want to handle linguistically relevant strings from multiple cultures in a consistent way. For example, if your application displays words that use multiple character sets in a list box, you might want to display words in the same order regardless of the current culture. For culture-

insensitive linguistic comparisons, .NET defines an invariant culture that is based on the linguistic conventions of English. To perform a culture-insensitive linguistic comparison, specify [StringComparison.InvariantCulture](#) or [StringComparison.InvariantCultureIgnoreCase](#) as the `comparisonType` parameter.

The following example performs two culture-insensitive, non-linguistic string comparisons. The first is case-sensitive, but the second is not.

C#

```
using System;

public class CompareSample
{
    public static void Main()
    {
        string string1 = "file";
        string string2 = "FILE";
        int compareResult = 0;

        compareResult = String.Compare(string1, string2,
                                       StringComparison.Ordinal);
        Console.WriteLine("{0} comparison of '{1}' and '{2}': {3}",
                          StringComparison.Ordinal, string1, string2,
                          compareResult);

        compareResult = String.Compare(string1, string2,
                                       StringComparison.OrdinalIgnoreCase);
        Console.WriteLine("{0} comparison of '{1}' and '{2}': {3}",
                          StringComparison.OrdinalIgnoreCase, string1,
                          string2,
                          compareResult);
    }
}

// The example displays the following output:
//   Ordinal comparison of 'file' and 'FILE': 32
//   OrdinalIgnoreCase comparison of 'file' and 'FILE': 0
```

You can download the [Sorting Weight Tables](#), a set of text files that contain information on the character weights used in sorting and comparison operations for Windows operating systems, and the [Default Unicode Collation Element Table](#), the sort weight table for Linux and macOS.

See also

- [String.Compare](#)
- [String.CompareTo](#)
- [Perform culture-insensitive string operations](#)

- Best practices for using strings

 **Collaborate with us on GitHub**

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

 **.NET feedback**

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

Perform culture-insensitive case changes

Article • 03/13/2023

The `String.ToUpper`, `String.ToLower`, `Char.ToUpper`, and `Char.ToLower` methods provide overloads that do not accept any parameters. By default, these overloads without parameters perform case changes based on the value of the `CultureInfo.CurrentCulture`. This produces case-sensitive results that can vary by culture. To make it clear whether you want case changes to be culture-sensitive or culture-insensitive, you should use the overloads of these methods that require you to explicitly specify a `culture` parameter. For culture-sensitive case changes, specify `CultureInfo.CurrentCulture` for the `culture` parameter. For culture-insensitive case changes, specify `CultureInfo.InvariantCulture` for the `culture` parameter.

Often, strings are converted to a standard case to enable easier lookup later. When strings are used in this way, you should specify `CultureInfo.InvariantCulture` for the `culture` parameter, because the value of `Thread.CurrentCulture` can potentially change between the time that the case is changed and the time that the lookup occurs.

If a security decision is based on a case change operation, the operation should be culture-insensitive to ensure that the result is not affected by the value of `CultureInfo.CurrentCulture`. See the "String Comparisons that Use the Current Culture" section of the [Best Practices for Using Strings](#) article for an example that demonstrates how culture-sensitive string operations can produce inconsistent results.

String.ToUpper and String.ToLower

For code clarity, it's recommended that you always use overloads of the `String.ToUpper` and `String.ToLower` methods that let you specify a culture explicitly. For example, the following code performs an identifier lookup. The `key.ToLower` operation is culture-sensitive by default, but this behavior is not clear from reading the code.

Example

C#

```
static object LookupKey(string key)
{
```

```
        return internalHashtable[key.ToLower()];
    }
```

If you want the `key.ToLower` operation to be culture-insensitive, change the preceding example as follows to explicitly use `CultureInfo.InvariantCulture` when changing the case.

C#

```
static object LookupKey(string key)
{
    return internalHashtable[key.ToLower(CultureInfo.InvariantCulture)];
}
```

Char.ToUpper and Char.ToLower

Although the `Char.ToUpper` and `Char.ToLower` methods have the same characteristics as the `String.ToUpper` and `String.ToLower` methods, the only cultures that are affected are Turkish (Türkiye) and Azerbaijani (Latin, Azerbaijan). These are the only two cultures with single-character casing differences. For more details about this unique case mapping, see the "Casing" section in the [String](#) class documentation. For code clarity and to ensure consistent results, it's recommended that you always use the overloads of these methods that accept a `CultureInfo` parameter.

See also

- [String.ToUpper](#)
- [String.ToLower](#)
- [Char.ToUpper](#)
- [Char.ToLower](#)
- [CA1311: Specify a culture or use an invariant version](#)
- [Change case in .NET](#)
- [Perform culture-insensitive string operations](#)

 Collaborate with us on
GitHub

The source for this content can
be found on GitHub, where you
can also create and review

.NET

.NET feedback

The .NET documentation is open
source. Provide feedback here.

issues and pull requests. For more information, see [our contributor guide](#).

 Open a documentation issue

 Provide product feedback

Perform culture-insensitive string operations in collections

Article • 03/18/2023

There are classes and members in the [System.Collections](#) namespace that provide culture-sensitive behavior by default. The parameterless constructors for the [CaseInsensitiveComparer](#) and [CaseInsensitiveHashCodeProvider](#) classes initialize a new instance using the [Thread.CurrentCulture](#) property. All overloads of the [CollectionsUtil.CreateCaseInsensitiveHashtable](#) method create a new instance of the [Hashtable](#) class using the [Thread.CurrentCulture](#) property by default. Overloads of the [ArrayList.Sort](#) method perform culture-sensitive sorts by default using [Thread.CurrentCulture](#). Sorting and lookup in a [SortedList](#) can be affected by [Thread.CurrentCulture](#) when strings are used as the keys. Follow the usage recommendations provided in this section to obtain culture-insensitive results from these classes and methods in the [collections](#) namespace.

ⓘ Note

Passing [CultureInfo.InvariantCulture](#) to a comparison method does perform a culture-insensitive comparison. However, it does not cause a non-linguistic comparison, for example, for file paths, registry keys, and environment variables. Neither does it support security decisions based on the comparison result. For a non-linguistic comparison or support for result-based security decisions, the application should use a comparison method that accepts a [StringComparison](#) value. The application should then pass [StringComparison](#).

Use the [CaseInsensitiveComparer](#) and [CaseInsensitiveHashCodeProvider](#) classes

The parameterless constructors for [CaseInsensitiveHashCodeProvider](#) and [CaseInsensitiveComparer](#) initialize a new instance of the class using the [Thread.CurrentCulture](#), resulting in culture-sensitive behavior. The following code example demonstrates the constructor for a [Hashtable](#) that is culture-sensitive because it uses the parameterless constructors for [CaseInsensitiveHashCodeProvider](#) and [CaseInsensitiveComparer](#).

C#

```
internalHashtable = new Hashtable(CaseInsensitiveHashCodeProvider.Default,
CaseInsensitiveComparer.Default);
```

If you want to create a culture-insensitive `Hashtable` using the `CaseInsensitiveComparer` and `CaseInsensitiveHashCodeProvider` classes, initialize new instances of these classes using the constructors that accept a `culture` parameter. For the `culture` parameter, specify `CultureInfo.InvariantCulture`. The following code example demonstrates the constructor for a culture-insensitive `Hashtable`.

C#

```
internalHashtable = new Hashtable(new CaseInsensitiveHashCodeProvider(
    CultureInfo.InvariantCulture),
    new CaseInsensitiveComparer(CultureInfo.InvariantCulture));
```

Use the `CollectionsUtil.CreateCaseInsensitiveHashTable` method

The `CollectionsUtil.CreateCaseInsensitiveHashTable` method is a useful shortcut for creating a new instance of the `Hashtable` class that ignores the case of strings. However, all overloads of the `CollectionsUtil.CreateCaseInsensitiveHashTable` method are culture-sensitive because they use the `Thread.CurrentCulture` property. You cannot create a culture-insensitive `Hashtable` using this method. To create a culture-insensitive `Hashtable`, use the `Hashtable` constructor that accepts a `culture` parameter. For the `culture` parameter, specify `CultureInfo.InvariantCulture`. The following code example demonstrates the constructor for a culture-insensitive `Hashtable`.

C#

```
internalHashtable = new Hashtable(new CaseInsensitiveHashCodeProvider(
    CultureInfo.InvariantCulture),
    new CaseInsensitiveComparer(CultureInfo.InvariantCulture));
```

Use the `SortedList` class

A `SortedList` represents a collection of key-and-value pairs that are sorted by the keys and are accessible by key and by index. When you use a `SortedList` where strings are

the keys, the sorting and lookup can be affected by the `Thread.CurrentCulture` property. To obtain culture-insensitive behavior from a `SortedList`, create a `SortedList` using one of the constructors that accepts a `comparer` parameter. The `comparer` parameter specifies the `IComparer` implementation to use when comparing keys. For the parameter, specify a custom comparer class that uses `CultureInfo.InvariantCulture` to compare keys. The following example illustrates a custom culture-insensitive comparer class that you can specify as the `comparer` parameter to a `SortedList` constructor.

C#

```
using System;
using System.Collections;
using System.Globalization;

internal class InvariantComparer : IComparer
{
    private CompareInfo _compareInfo;
    internal static readonly InvariantComparer Default = new
        InvariantComparer();

    internal InvariantComparer()
    {
        _compareInfo = CultureInfo.InvariantCulture.CompareInfo;
    }

    public int Compare(Object a, Object b)
    {
        if (a is string sa && b is string sb)
            return _compareInfo.Compare(sa, sb);
        else
            return Comparer.Default.Compare(a,b);
    }
}
```

In general, if you use a `SortedList` on strings without specifying a custom invariant comparer, a change to `Thread.CurrentCulture` after the list has been populated can invalidate the list.

Use the `ArrayList.Sort` method

Overloads of the `ArrayList.Sort` method perform culture-sensitive sorts by default using the `Thread.CurrentCulture` property. Results can vary by culture due to different sort orders. To eliminate culture-sensitive behavior, use the overloads of this method that accept an `IComparer` implementation. For the `comparer` parameter, specify a custom

invariant comparer class that uses `CultureInfo.InvariantCulture`. An example of a custom invariant comparer class is provided in the [Using the SortedList Class](#) topic.

See also

- [CaseInsensitiveComparer](#)
- [CaseInsensitiveHashCodeProvider](#)
- [ArrayList.Sort](#)
- [SortedList](#)
- [Hashtable](#)
- [IComparer](#)
- [Perform culture-insensitive string operations](#)
- [CollectionsUtil.CreateCaseInsensitiveHashtable](#)

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

 .NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

Perform culture-insensitive string operations in arrays

Article • 03/18/2023

Overloads of the [Array.Sort](#) and [Array.BinarySearch](#) methods perform culture-sensitive sorts by default using the [Thread.CurrentCulture](#) property. Culture-sensitive results returned by these methods can vary by culture due to differences in sort orders. To eliminate culture-sensitive behavior, use one of the overloads of this method that accepts a `comparer` parameter. The `comparer` parameter specifies the [IComparer](#) implementation to use when comparing elements in the array. For the parameter, specify a custom invariant comparer class that uses [CultureInfo.InvariantCulture](#). An example of a custom invariant comparer class is provided in the "Using the SortedList Class" subtopic of the [Perform culture-insensitive string operations in collections](#) topic.

ⓘ Note

Passing [CultureInfo.InvariantCulture](#) to a comparison method does perform a culture-insensitive comparison. However, it does not cause a non-linguistic comparison, for example, for file paths, registry keys, and environment variables. Neither does it support security decisions based on the comparison result. For a non-linguistic comparison or support for result-based security decisions, the application should use a comparison method that accepts a [StringComparison](#) value. The application should then pass [Ordinal](#).

See also

- [Array.Sort](#)
- [Array.BinarySearch](#)
- [IComparer](#)
- [Perform culture-insensitive string operations](#)

ⓘ Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

ⓘ Open a documentation issue

issues and pull requests. For more information, see [our contributor guide](#).

 [Provide product feedback](#)

Best practices for developing world-ready applications

Article • 03/18/2023

This section describes the best practices to follow when developing world-ready applications.

Globalization best practices

1. Make your application Unicode internally.
2. Use the culture-aware classes provided by the [System.Globalization](#) namespace to manipulate and format data.
 - For sorting, use the [SortKey](#) class and the [CompareInfo](#) class.
 - For string comparisons, use the [CompareInfo](#) class.
 - For date and time formatting, use the [DateTimeFormatInfo](#) class.
 - For numeric formatting, use the [NumberFormatInfo](#) class.
 - For Gregorian and non-Gregorian calendars, use the [Calendar](#) class or one of the specific calendar implementations.
3. Use the culture property settings provided by the [System.Globalization.CultureInfo](#) class in the appropriate situations. Use the [CultureInfo.CurrentCulture](#) property for formatting tasks, such as date and time or numeric formatting. Use the [CultureInfo.CurrentUICulture](#) property to retrieve resources. Note that the `CurrentCulture` and `CurrentUICulture` properties can be set per thread.
4. Enable your application to read and write data to and from a variety of encodings by using the encoding classes in the [System.Text](#) namespace. Do not assume ASCII data. Assume that international characters will be supplied anywhere a user can enter text. For example, the application should accept international characters in server names, directories, file names, user names, and URLs.
5. When using the [UTF8Encoding](#) class, for security reasons, use the error detection feature offered by this class. To turn on the error detection feature, create an instance of the class using the constructor that takes a `throwOnInvalidBytes` parameter and set the value of this parameter to `true`.
6. Whenever possible, handle strings as entire strings instead of as a series of individual characters. This is especially important when sorting or searching for

substrings. This will prevent problems associated with parsing combined characters. You can also work with units of text rather than single characters by using the [System.Globalization.StringInfo](#) class.

7. Display text using the classes provided by the [System.Drawing](#) namespace.
8. For consistency across operating systems, do not allow user settings to override [CultureInfo](#). Use the `cultureInfo` constructor that accepts a `useUserOverride` parameter and set it to `false`.
9. Test your application functionality on international operating system versions, using international data.
10. If a security decision is based on the result of a string comparison or case change operation, use a culture-insensitive string operation. This practice ensures that the result is not affected by the value of `cultureInfo.CurrentCulture`. See the "[String Comparisons that Use the Current Culture](#)" section of [Best Practices for Using Strings](#) for an example that demonstrates how culture-sensitive string comparisons can produce inconsistent results.

Localization best practices

1. Move all localizable resources to separate resource-only DLLs. Localizable resources include user interface elements, such as strings, error messages, dialog boxes, menus, and embedded object resources.
2. Do not hardcode strings or user interface resources.
3. Do not put non-localizable resources into the resource-only DLLs. This confuses translators.
4. Do not use composite strings that are built at run time from concatenated phrases. Composite strings are difficult to localize because they often assume an English grammatical order that does not apply to all languages.
5. Avoid ambiguous constructs such as "Empty Folder" where the strings can be translated differently depending on the grammatical roles of the string components. For example, "empty" can be either a verb or an adjective, which can lead to different translations in languages such as Italian or French.
6. Avoid using images and icons that contain text in your application. They are expensive to localize.

7. Allow plenty of room for the length of strings to expand in the user interface. In some languages, phrases can require 50-75 percent more space than they need in other languages.
8. Use the [System.Resources.ResourceManager](#) class to retrieve resources based on culture.
9. Use [Visual Studio](#) to create Windows Forms dialog boxes so they can be localized using the [Windows Forms Resource Editor \(Winres.exe\)](#). Do not code Windows Forms dialog boxes by hand.
10. Arrange for professional localization (translation).
11. For a complete description of creating and localizing resources, see [Resources in .NET apps](#).

Globalization best practices for ASP.NET applications

Tip

The following best practices are for ASP.NET Framework apps. For ASP.NET Core apps, see [Globalization and localization in ASP.NET Core](#).

1. Explicitly set the [CurrentUICulture](#) and [CurrentCulture](#) properties in your application. Do not rely on defaults.
2. Note that ASP.NET applications are managed applications and therefore can use the same classes as other managed applications for retrieving, displaying, and manipulating information based on culture.
3. Be aware that you can specify the following three types of encodings in ASP.NET:
 - `requestEncoding` specifies the encoding received from the client's browser.
 - `responseEncoding` specifies the encoding to send to the client browser. In most situations, this encoding should be the same as that specified for `requestEncoding`.
 - `fileEncoding` specifies the default encoding for `.aspx`, `.asmx`, and `.asax` file parsing.
4. Specify the values for the `requestEncoding`, `responseEncoding`, `fileEncoding`, `culture`, and `uiCulture` attributes in the following three places in an ASP.NET

application:

- In the globalization section of a `Web.config` file. This file is external to the ASP.NET application. For more information, see [`<globalization>` element](#).
- In a page directive. Note that, when an application is in a page, the file has already been read. Therefore, it is too late to specify `fileEncoding` and `requestEncoding`. Only `uiCulture`, `culture`, and `responseEncoding` can be specified in a page directive.
- Programmatically in application code. This setting can vary per request. As with a page directive, by the time the application's code is reached, it is too late to specify `fileEncoding` and `requestEncoding`. Only `uiCulture`, `culture`, and `responseEncoding` can be specified in the application code.

5. Note that the `uiCulture` value can be set to the browser accept language.

See also

- [Globalization and Localization](#)
- [Resources in .NET apps](#)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

Localizability review

Article • 03/18/2023

The localizability review is an intermediate step in the development of a world-ready application. It verifies that a globalized application is ready for localization and identifies any code or any aspects of the user interface that require special handling. This step also helps ensure that the localization process will not introduce any functional defects into your application. When all the issues raised by the localizability review have been addressed, your application is ready for localization. If the localizability review is thorough, you should not have to modify any source code during the localization process.

The localizability review consists of the following three checks:

- ✓ Are the globalization recommendations implemented?
- ✓ Are culture-sensitive features handled correctly?
- ✓ Have you tested your application with international data?

Implement globalization recommendations

If you have designed and developed your application with localization in mind, and if you have followed the recommendations discussed in the [Globalization](#) article, the localizability review will largely be a quality assurance pass. Otherwise, during this stage you should review and implement the recommendations for [globalization](#) and fix the errors in source code that prevent localization.

Handle culture-sensitive features

.NET does not provide programmatic support in a number of areas that vary widely by culture. In most cases, you have to write custom code to handle feature areas like the following:

- Addresses
- Telephone numbers
- Paper sizes
- Units of measure used for lengths, weights, area, volume, and temperatures

Although .NET does not offer built-in support for converting between units of measure, you can use the `RegionInfo.IsMetric` property to determine whether a particular country or region uses the metric system, as the following example illustrates.

C#

```
using System;
using System.Globalization;

public class Example
{
    public static void Main()
    {
        string[] cultureNames = { "en-US", "en-GB", "fr-FR",
                                  "ne-NP", "es-BO", "ig-NG" };
        foreach (var cultureName in cultureNames) {
            RegionInfo region = new RegionInfo(cultureName);
            Console.WriteLine("{0} {1} the metric system.",
region.EnglishName,
                           region.IsMetric ? "uses" : "does not use");
        }
    }
}
// The example displays the following output:
//      United States does not use the metric system.
//      United Kingdom uses the metric system.
//      France uses the metric system.
//      Nepal uses the metric system.
//      Bolivia uses the metric system.
//      Nigeria uses the metric system.
```

Test your application

Before you localize your application, you should test it by using international data on international versions of the operating system. Although most of the user interface will not be localized at this point, you will be able to detect problems such as the following:

- Serialized data that does not deserialize correctly across operating system versions.
- Numeric data that does not reflect the conventions of the current culture. For example, numbers may be displayed with inaccurate group separators, decimal separators, or currency symbols.
- Date and time data that does not reflect the conventions of the current culture. For example, numbers that represent the month and day may appear in the wrong

order, date separators may be incorrect, or time zone information may be incorrect.

- Resources that cannot be found because you have not identified a default culture for your application.
- Strings that are displayed in an unusual order for the specific culture.
- String comparisons or comparisons for equality that return unexpected results.

If you've followed the globalization recommendations when developing your application, handled culture-sensitive features correctly, and identified and addressed the localization issues that arose during testing, you can proceed to the next step, [Localization](#).

See also

- [Globalization and Localization](#)
- [Localization](#)
- [Globalization](#)
- [Resources in .NET apps](#)

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

Localization in .NET

Article • 12/19/2023

Localization is the process of translating an application's resources into localized versions for each culture that the application will support. You should proceed to the localization step only after completing the [Localizability review](#) step to verify that the globalized application is ready for localization.

An application that is ready for localization is separated into two conceptual blocks: a block that contains all user interface elements and a block that contains executable code. The user interface block contains only localizable user-interface elements such as strings, error messages, dialog boxes, menus, embedded object resources, and so on for the neutral culture. The code block contains only the application code to be used by all supported cultures. The common language runtime supports a satellite assembly resource model that separates an application's executable code from its resources. For more information about implementing this model, see [Resources in .NET](#).

For each localized version of your application, add a new satellite assembly that contains the localized user interface block translated into the appropriate language for the target culture. The code block for all cultures should remain the same. The combination of a localized version of the user interface block with the code block produces a localized version of your application.

In this article, you will learn how to use the [IStringLocalizer<T>](#) and [IStringLocalizerFactory](#) implementations. All of the example source code in this article relies on the [Microsoft.Extensions.Localization](#) ↗ and [Microsoft.Extensions.Hosting](#) ↗ NuGet packages. For more information on hosting, see [.NET Generic Host](#).

Resource files

The primary mechanism for isolating localizable strings is with **resource files**. A resource file is an XML file with the `.resx` file extension. Resource files are translated prior to the execution of the consuming application — in other words, they represent translated content at rest. A resource file name most commonly contains a locale identifier, and takes on the following form:

`<FullTypeName><.Locale>.resx`

Where:

- The `<FullTypeName>` represents localizable resources for a specific type.

- The optional `<.Locale>` represents the locale of the resource file contents.

Specifying locales

The locale should define the language, at a bare minimum, but it can also define the culture (regional language), and even the country or region. These segments are commonly delimited by the `-` character. With the added specificity of a culture, the "culture fallback" rules are applied where best matches are prioritized. The locale should map to a well-known language tag. For more information, see [CultureInfo.Name](#).

Culture fallback scenarios

Imagine that your localized app supports various Serbian locales, and has the following resource files for its `MessageService`:

[\[+\] Expand table](#)

| File | Regional language | Country Code |
|---|-------------------------------|--------------|
| <code>MessageService.sr-Cyrl-RS.resx</code> | (Cyrillic, Serbia) | RS |
| <code>MessageService.sr-Cyrl.resx</code> | Cyrillic | |
| <code>MessageService.sr-Latn-BA.resx</code> | (Latin, Bosnia & Herzegovina) | BA |
| <code>MessageService.sr-Latn-ME.resx</code> | (Latin, Montenegro) | ME |
| <code>MessageService.sr-Latn-RS.resx</code> | (Latin, Serbia) | RS |
| <code>MessageService.sr-Latn.resx</code> | Latin | |
| <code>MessageService.sr.resx</code> | [†] Latin | |
| <code>MessageService.resx</code> | | |

[†] The default regional language for the language.

When your app is running with the `CultureInfo.CurrentCulture` set to a culture of `"sr-Cyrl-RS"` localization attempts to resolve files in the following order:

1. `MessageService.sr-Cyrl-RS.resx`
2. `MessageService.sr-Cyrl.resx`
3. `MessageService.sr.resx`
4. `MessageService.resx`

However, if your app was running with the `CultureInfo.CurrentCulture` set to a culture of `"sr-Latn-BA"` localization attempts to resolve files in the following order:

1. `MessageService.sr-Latn-BA.resx`
2. `MessageService.sr-Latn.resx`
3. `MessageService.sr.resx`
4. `MessageService.resx`

The "culture fallback" rule will ignore locales when there are no corresponding matches, meaning resource file number four is selected if it's unable to find a match. If the culture was set to `"fr-FR"`, localization would end up falling to the `MessageService.resx` file which can be problematic. For more information, see [The resource fallback process](#).

Resource lookup

Resource files are automatically resolved as part of a lookup routine. If your project file name is different than the root namespace of your project, the assembly name might differ. This can prevent resource lookup from being otherwise successful. To address this mismatch, use the `RootNamespaceAttribute` to provide a hint to the localization services. When provided, it is used during resource lookup.

The example project is named `example.csproj`, which creates an `example.dll` and `example.exe` — however, the `Localization.Example` namespace is used. Apply an `assembly` level attribute to correct this mismatch:

```
C#  
[assembly: RootNamespace("Localization.Example")]
```

Register localization services

To register localization services, call one of the `AddLocalization` extension methods during the configuration of services. This will enable dependency injection (DI) of the following types:

- `Microsoft.Extensions.Localization.IStringLocalizer<T>`
- `Microsoft.Extensions.Localization.IStringLocalizerFactory`

Configure localization options

The `AddLocalization(IServiceCollection, Action<LocalizationOptions>)` overload accepts a `setupAction` parameter of type `Action<LocalizationOptions>`. This allows you to configure localization options.

C#

```
HostApplicationBuilder builder = Host.CreateApplicationBuilder(args);

builder.Services.AddLocalization(options =>
{
    options.ResourcesPath = "Resources";
});

// Omitted for brevity.
```

Resource files can live anywhere in a project, but there are common practices in place that have proven to be successful. More often than not, the path of least resistance is followed. The preceding C# code:

- Creates the default host app builder.
- Calls `AddLocalization` on the service collection, specifying `LocalizationOptions.ResourcesPath` as `"Resources"`.

This would cause the localization services to look in the `Resources` directory for resource files.

Use `IStringLocalizer<T>` and `IStringLocalizerFactory`

After you've [registered](#) (and optionally [configured](#)) the localization services, you can use the following types with DI:

- `IStringLocalizer<T>`
- `IStringLocalizerFactory`

To create a message service that is capable of returning localized strings, consider the following `MessageService`:

C#

```
using System.Diagnostics.CodeAnalysis;
using Microsoft.Extensions.Localization;

namespace Localization.Example;
```

```

public sealed class MessageService(IStringLocalizer<MessageService>
localizer)
{
    [return: NotNullIfNotNull(nameof(localizer))]
    public string? GetGreetingMessage()
    {
        LocalizedString localizedString = localizer["GreetingMessage"];

        return localizedString;
    }
}

```

In the preceding C# code:

- A `IStringLocalizer<MessageService>` `localizer` field is declared.
- The primary constructor defines an `IStringLocalizer<MessageService>` parameter and captures it as a `localizer` argument.
- The `GetGreetingMessage` method invokes the `IStringLocalizer.Item[String]` passing `"GreetingMessage"` as an argument.

The `IStringLocalizer` also supports parameterized string resources, consider the following `ParameterizedMessageService`:

C#

```

using System.Diagnostics.CodeAnalysis;
using Microsoft.Extensions.Localization;

namespace Localization.Example;

public class ParameterizedMessageService(IStringLocalizerFactory factory)
{
    private readonly IStringLocalizer _localizer =
        factory.Create(typeof(ParameterizedMessageService));

    [return: NotNullIfNotNull(nameof(_localizer))]
    public string? GetFormattedMessage(DateTime dateTime, double
dinnerPrice)
    {
        LocalizedString localizedString = _localizer["DinnerPriceFormat",
dateTime, dinnerPrice];

        return localizedString;
    }
}

```

In the preceding C# code:

- A `IStringLocalizer _localizer` field is declared.

- The primary constructor takes an `IStringLocalizerFactory` parameter, which is used to create an `IStringLocalizer` from the `ParameterizedMessageService` type, and assigns it to the `_localizer` field.
- The `GetFormattedMessage` method invokes `IStringLocalizer.Item[String, Object[]]`, passing `"DinnerPriceFormat"`, a `dateTime` object, and `dinnerPrice` as arguments.

ⓘ Important

The `IStringLocalizerFactory` isn't required. Instead, it is preferred for consuming services to require the `IStringLocalizer<T>`.

Both `IStringLocalizer.Item[]` indexers return a `LocalizedString`, which have `implicit conversions` to `string?`.

Put it all together

To exemplify an app using both message services, along with localization and resource files, consider the following `Program.cs` file:

C#

```
using System.Globalization;
using Localization.Example;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
using Microsoft.Extensions.Localization;
using Microsoft.Extensions.Logging;
using static System.Console;
using static System.Text.Encoding;

[assembly: RootNamespace("Localization.Example")]

OutputEncoding = Unicode;

if (args is { Length: 1 })
{
    CultureInfo.CurrentCulture =
        CultureInfo.CurrentUICulture =
            CultureInfo.GetCultureInfo(args[0]);
}

HostApplicationBuilder builder = Host.CreateApplicationBuilder(args);

builder.Services.AddLocalization();
builder.Services.AddTransient<MessageService>();
builder.Services.AddTransient<ParameterizedMessageService>();
builder.Logging.SetMinimumLevel(LogLevel.Warning);
```

```

using IHost host = builder.Build();

IServiceProvider services = host.Services;

ILogger logger =
    services.GetRequiredService<ILoggerFactory>()
        .CreateLogger("Localization.Example");

MessageService messageService =
    services.GetRequiredService<MessageService>();
logger.LogWarning(
    "{Msg}",
    messageService.GetGreetingMessage());

ParameterizedMessageService parameterizedMessageService =
    services.GetRequiredService<ParameterizedMessageService>();
logger.LogWarning(
    "{Msg}",
    parameterizedMessageService.GetFormattedMessage(
        DateTime.Today.AddDays(-3), 37.63));

await host.RunAsync();

```

In the preceding C# code:

- The [RootNamespaceAttribute](#) sets "Localization.Example" as the root namespace.
- The [Console.OutputEncoding](#) is assigned to [Encoding.Unicode](#).
- When a single argument is passed to `args`, the [CultureInfo.CurrentCulture](#) and [CultureInfo.CurrentUICulture](#) are assigned the result of [CultureInfo.GetCultureInfo\(String\)](#) given the `arg[0]`.
- The [Host](#) is created with [defaults](#).
- The localization services, `MessageService`, and `ParameterizedMessageService` are registered to the `IServiceCollection` for DI.
- To remove noise, logging is configured to ignore any log level lower than a warning.
- The `MessageService` is resolved from the `IServiceProvider` instance and its resulting message is logged.
- The `ParameterizedMessageService` is resolved from the `IServiceProvider` instance and its resulting formatted message is logged.

Each of the `*MessageService` classes defines a set of `.resx` files, each with a single entry. Here is the example content for the `MessageService` resource files, starting with `MessageService.resx`:

XML

```
<?xml version="1.0" encoding="utf-8"?>
<root>
  <data name="GreetingMessage" xml:space="preserve">
    <value>Hi friends, the ".NET" developer community is excited to see you here!</value>
  </data>
</root>
```

MessageService.sr-Cyrl-RS.resx:

XML

```
<?xml version="1.0" encoding="utf-8"?>
<root>
  <data name="GreetingMessage" xml:space="preserve">
    <value>Здраво пријатељи, ".NET" девелопер заједница је узбуђена што вас види овде!</value>
  </data>
</root>
```

MessageService.sr-Latn.resx:

XML

```
<?xml version="1.0" encoding="utf-8"?>
<root>
  <data name="GreetingMessage" xml:space="preserve">
    <value>Zdravo prijatelji, ".NET" developer zajednica je uzbudena sto vas vidi ovde!</value>
  </data>
</root>
```

Here is the example content for the `ParameterizedMessageService` resource files, starting with `ParameterizedMessageService.resx`:

XML

```
<?xml version="1.0" encoding="utf-8"?>
<root>
  <data name="DinnerPriceFormat" xml:space="preserve">
    <value>On {0:D} my dinner cost {1:C}.</value>
  </data>
</root>
```

ParameterizedMessageService.sr-Cyrl-RS.resx:

XML

```
<?xml version="1.0" encoding="utf-8"?>
<root>
    <data name="DinnerPriceFormat" xml:space="preserve">
        <value>У {0:D} моја вечера је коштала {1:C}.</value>
    </data>
</root>
```

ParameterizedMessageService.sr-Latn.resx:

XML

```
<?xml version="1.0" encoding="utf-8"?>
<root>
    <data name="DinnerPriceFormat" xml:space="preserve">
        <value>U {0:D} moja večera je koštala {1:C}.</value>
    </data>
</root>
```

Tip

All of the resource file XML comments, schema, and `<resheader>` elements are intentionally omitted for brevity.

Example runs

The following example runs show the various localized outputs, given targeted locales.

Consider "sr-Latn":

.NET CLI

```
dotnet run --project .\example\example.csproj sr-Latn

warn: Localization.Example[0]
      Zdravo prijatelji, ".NET" developer zajednica je uzbudena što vas vidi
ovde!
warn: Localization.Example[0]
      U utorak, 03. avgust 2021. moja večera je koštala 37,63 ₾.
```

When omitting an argument to the .NET CLI to run the project, the default system culture is used — in this case "en-US":

.NET CLI

```
dotnet run --project .\example\example.csproj

warn: Localization.Example[0]
      Hi friends, the ".NET" developer community is excited to see you here!
warn: Localization.Example[0]
      On Tuesday, August 3, 2021 my dinner cost $37.63.
```

When passing "sr-Cryl-RS", the correct corresponding resource files are found and the localization applied:

```
.NET CLI

dotnet run --project .\example\example.csproj sr-Cryl-RS

warn: Localization.Example[0]
      Здраво пријатељи, ".NET" девелопер заједница је узбуђена што вас види
овде!
warn: Localization.Example[0]
      У уторак, 03. август 2021. моја вечера је коштала 38 RSD.
```

The sample application does not provide resource files for "fr-CA", but when called with that culture, the non-localized resource files are used.

⚠ Warning

Since the culture is found but the correct resource files are not, when formatting is applied you end up with partial localization:

```
.NET CLI

dotnet run --project .\example\example.csproj fr-CA

warn: Localization.Example[0]
      Hi friends, the ".NET" developer community is excited to see you
here!
warn: Localization.Example[0]
      On mardi 3 août 2021 my dinner cost 37,63 $.
```

See also

- [Globalizing and localizing .NET applications](#)
- [Package and deploy resources in .NET apps](#)
- [Microsoft.Extensions.Localization ↗](#)

- Dependency injection in .NET
- Logging in .NET
- ASP.NET Core localization

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

.NET is an open source project.
Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

CompareInfo class

Article • 12/30/2023

This article provides supplementary remarks to the reference documentation for this API.

Conventions for comparing and sorting data vary from culture to culture. For example, sort order may be based on phonetics or on the visual representation of characters. In East Asian languages, characters are sorted by the stroke and radical of ideographs. Sorting also depends on the order languages and cultures use for the alphabet. For example, the Danish language has an "Æ" character that it sorts after "Z" in the alphabet. In addition, comparisons may be case-sensitive or case-insensitive, and casing rules may also differ by culture. The [CompareInfo](#) class is responsible for maintaining this culture-sensitive string comparison data and for performing culture-sensitive string operations.

Typically, you do not have to instantiate a [CompareInfo](#) object directly, because one is used implicitly by all non-ordinal string comparison operations, including calls to the [String.Compare](#) method. However, if you do want to retrieve a [CompareInfo](#) object, you can do it in one of these ways:

- By retrieving the value of the [CultureInfo.CompareInfo](#) property for a particular culture.
- By calling the static [GetCompareInfo](#) method with a culture name. This allows for late-bound access to a [CompareInfo](#) object.

Ignored search values

Character sets include ignorable characters, which are characters that are not considered when performing a linguistic or culture-sensitive comparison. Comparison methods such as [IndexOf](#) and [LastIndexOf](#) do not consider such characters when they perform a culture-sensitive comparison. Ignorable characters include:

- [String.Empty](#). Culture-sensitive comparison methods will always find an empty string at the beginning (index zero) of the string being searched.
- A character or string consisting of characters with code points that are not considered in the operation because of comparison options. In particular, the [CompareOptions.IgnoreNonSpace](#) and [CompareOptions.IgnoreSymbols](#) options produce searches in which symbols and nonspacing combining characters are ignored.

- A string with code points that have no linguistic significance. For example, a soft hyphen (U+00AD) is always ignored in a culture-sensitive string comparison.

Security considerations

If a security decision depends on a string comparison or a case change, you should use the [InvariantCulture](#) property to ensure that the behavior is consistent, regardless of the culture settings of the operating system.

Note

When possible, you should use string comparison methods that have a parameter of type [CompareOptions](#) to specify the kind of comparison expected. As a general rule, use linguistic options (using the current culture) for comparing strings displayed in the user interface and specify [Ordinal](#) or [OrdinalIgnoreCase](#) for security comparisons.

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

.NET is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

CompareOptions enum

Article • 12/30/2023

This article provides supplementary remarks to the reference documentation for this API.

The [CompareOptions](#) options denote case sensitivity or necessity to ignore types of characters.

.NET uses three distinct ways of sorting: word sort, string sort, and ordinal sort. Word sort performs a culture-sensitive comparison of strings. Certain nonalphanumeric characters might have special weights assigned to them. For example, the hyphen ("–") might have a very small weight assigned to it so that "coop" and "co-op" appear next to each other in a sorted list. String sort is similar to word sort, except that there are no special cases. Therefore, all nonalphanumeric symbols come before all alphanumeric characters. Ordinal sort compares strings based on the Unicode values of each element of the string. For a downloadable set of text files that contain information on the character weights used in sorting and comparison operations for Windows operating systems, see [Sorting Weight Tables](#). For the sort weight table for Linux and macOS, see the [Default Unicode Collation Element Table](#). The specific version of the sort weight table on Linux and macOS depends on the version of the [International Components for Unicode](#) libraries installed on the system. For information on ICU versions and the Unicode versions that they implement, see [Downloading ICU](#).

The `StringSort` value can only be used with [CompareInfo.Compare](#) and [CompareInfo.GetSortKey](#). [ArgumentException](#) is thrown if the `StringSort` value is used with [CompareInfo.IsPrefix](#), [CompareInfo.IsSuffix](#), [CompareInfo.IndexOf](#), or [CompareInfo.LastIndexOf](#).

ⓘ Note

When possible, you should use string comparison methods that accept a [CompareOptions](#) value to specify the kind of comparison expected. As a general rule, user-facing comparisons are best served by the use of linguistic options (using the current culture), while security comparisons should specify `ordinal` or `OrdinalIgnoreCase`.

Culture-sensitive sorts

Note

.NET Core running on Linux and macOS systems only: The collation behavior for the C and Posix cultures is always case-sensitive because these cultures do not use the expected Unicode collation order. We recommend that you use a culture other than C or Posix for performing culture-sensitive, case-insensitive sorting operations.

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

.NET is an open source project.
Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

CultureAndRegionInfoBuilder class

Article • 01/10/2024

This article provides supplementary remarks to the reference documentation for this API.

ⓘ Note

The [CultureAndRegionInfoBuilder](#) class is useful only for Windows operating systems. The generated *.nlp* files aren't supported on non-Windows operating systems. Also, even on Windows, the generated *.nlp* files are supported only on .NET Framework (or in .NET Core when using [NLS globalization mode](#)).

The [CultureInfo](#) class holds culture-specific information, such as the associated language, sublanguage, country/region, calendar, and cultural conventions. This class also provides culture-specific instances of the [DateTimeFormatInfo](#), [NumberFormatInfo](#), [CompareInfo](#), and [TextInfo](#) classes, which are required for culture-specific operations such as casing, formatting and parsing dates and numbers, and comparing strings.

By default, .NET supports [CultureInfo](#) objects that represent a predefined set of cultures. For a list of these cultures available on Windows systems, see the [Language tag](#) column in the [list of language/region names supported by Windows](#). Culture names follow the standard defined by [BCP 47](#). The [CultureAndRegionInfoBuilder](#) class enables you to create a custom culture that is completely new or that overrides a predefined culture. When a custom culture is installed and registered on a particular computer, it becomes indistinguishable from predefined [CultureInfo](#) objects, and can be instantiated and used just like those objects.

ⓘ Important

The [CultureAndRegionInfoBuilder](#) class is found in an assembly named *sysglobl.dll*. To successfully compile code that uses this type, you must add a reference to *sysglobl.dll*.

A custom culture can be registered on a computer only by a user who has administrative rights on that computer. Consequently, apps typically do not create and install custom cultures. Instead, you can use the [CultureAndRegionInfoBuilder](#) class to create a special-purpose tool that an administrator can use to create, install, and register a custom culture. After the custom culture is registered on a computer, you can use the

`CultureInfo` class in your app to create instances of the custom culture just as you would for a predefined culture.

If you parse date and time strings generated for a custom culture, you should use the `DateTime.ParseExact` or `DateTime.TryParseExact` method instead of the `DateTime.Parse` or `DateTime.TryParse` method to improve the probability that the parse operation will succeed. A date and time string for a custom culture can be complicated and therefore difficult to parse. The `Parse` and `TryParse` methods try to parse a string with several implicit parse patterns, all of which might fail. The `TryParseExact` method, in contrast, requires the application to explicitly designate one or more exact parse patterns that are likely to succeed.

Define and create a custom culture

You use the `CultureAndRegionInfoBuilder` class to define and name a custom culture. The custom culture can be an entirely new culture, a new culture that is based on an existing culture (that is, a supplemental culture), or a culture that replaces an existing .NET culture. In each case, the basic steps are the same:

1. Instantiate a `CultureAndRegionInfoBuilder` object by calling its `CultureAndRegionInfoBuilder(String, CultureAndRegionModifiers)` constructor. To replace an existing culture, pass that culture's name and the `CultureAndRegionModifiers.Replacement` enumeration value to the constructor. To create a new culture or a supplemental culture, pass a unique culture name and either the `CultureAndRegionModifiers.Neutral` or `CultureAndRegionModifiers.None` enumeration value.

Note

If you use the `CultureAndRegionModifiers.Replacement` enumeration value to instantiate a `CultureAndRegionInfoBuilder` object, the `CultureAndRegionInfoBuilder` object's properties are automatically populated with values from the `CultureInfo` object to be replaced.

2. If you're creating a new or supplemental culture:

- Populate the `CultureAndRegionInfoBuilder` object's properties by calling the `LoadDataFromCultureInfo` method and passing a `CultureInfo` object whose property values are similar to your new object.
- Populate the `CultureAndRegionInfoBuilder` object's regional properties by calling the `LoadDataFromRegionInfo` method and passing a `RegionInfo`

object that represents the region of your custom culture.

3. Modify the properties of the [CultureAndRegionInfoBuilder](#) object as necessary.
4. If you are planning to register the custom culture in a separate routine, call the [Save](#) method. This generates an XML file that you can load and register in a separate custom culture installation routine.

Register a custom culture

If you are developing a registration application for a custom culture that is separate from the application that creates the culture, you call the [CreateFromLdml](#) method to load the XML file that contains the custom culture's definition and instantiate the [CultureAndRegionInfoBuilder](#) object. To handle the registration, call the [Register](#) method. For the registration to succeed, the application that registers the custom culture must be running with administrative privileges on the target system; otherwise, the call to [Register](#) throws an [UnauthorizedAccessException](#) exception.

Warning

Culture data can differ between systems. If you're using the [CultureAndRegionInfoBuilder](#) class to create a custom culture that is uniform across multiple systems and you are creating your custom culture by loading data from existing [CultureInfo](#) and [RegionInfo](#) objects and customizing it, you should develop two different utilities. The first creates the custom culture and saves it to an XML file. The second uses the [CreateFromLdml](#) method to load the custom culture from an XML file and register it on the target computer.

The registration process performs the following tasks:

- Creates an *.nlp* file that contains the information that's defined in the [CultureAndRegionInfoBuilder](#) object.
- Stores the *.nlp* file in the %windir%\Globalization system directory on the target computer. This enables the custom culture's settings to persist between sessions. (The [CultureAndRegionInfoBuilder](#) method requires administrative privileges because the *.nlp* file is stored in a system directory.)
- Prepares .NET to search the %windir%\Globalization system directory instead of an internal cache the next time there's a request to create your new custom culture.

When a custom culture is successfully registered, it's indistinguishable from the cultures that are predefined by .NET. The custom culture is available until a call to the

[CultureAndRegionInfoBuilder](#) method removes the *.nlp* file from the local computer.

Instantiate a custom culture

You can create an instance of the custom culture in one of the following ways:

- By invoking the [CultureInfo.CultureInfo](#) constructor with the culture name.
- By calling the [CultureInfo.CreateSpecificCulture](#) method with the culture name.
- By calling the [CultureInfo.GetCultureInfo](#) method with the culture name.

In addition, the array of [CultureInfo](#) objects that is returned by the [CultureInfo.GetCultures](#) method includes the custom culture.

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

.NET is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

CultureInfo class

Article • 12/30/2023

This article provides supplementary remarks to the reference documentation for this API.

The [CultureInfo](#) class provides culture-specific information, such as the language, sublanguage, country/region, calendar, and conventions associated with a particular culture. This class also provides access to culture-specific instances of the [DateTimeFormatInfo](#), [NumberFormatInfo](#), [CompareInfo](#), and [TextInfo](#) objects. These objects contain the information required for culture-specific operations, such as casing, formatting dates and numbers, and comparing strings. The [CultureInfo](#) class is used either directly or indirectly by classes that format, parse, or manipulate culture-specific data, such as [String](#), [DateTime](#), [DateTimeOffset](#), and the numeric types.

Culture names and identifiers

The [CultureInfo](#) class specifies a unique name for each culture, based on RFC 4646. The name is a combination of an ISO 639 two-letter or three-letter lowercase culture code associated with a language and an ISO 3166 two-letter uppercase subculture code associated with a country or region. In addition, for apps that target .NET Framework 4 or later and are running under Windows 10 or later, culture names that correspond to valid BCP-47 language tags are supported.

ⓘ Note

When a culture name is passed to a class constructor or a method such as [CreateSpecificCulture](#) or [CultureInfo](#), its case is not significant.

The format for the culture name based on RFC 4646 is `Languagecode2 - country/regioncode2`, where `Languagecode2` is the two-letter language code and `country/regioncode2` is the two-letter subculture code. Examples include `ja-JP` for Japanese (Japan) and `en-US` for English (United States). In cases where a two-letter language code is not available, a three-letter code as defined in ISO 639-3 is used.

Some culture names also specify an ISO 15924 script. For example, `Cyr` specifies the Cyrillic script and `Latn` specifies the Latin script. A culture name that includes a script uses the pattern `Languagecode2 - scripttag - country/regioncode2`. An example of this type of culture name is `uz-Cyr1-UZ` for Uzbek (Cyrillic, Uzbekistan). On Windows

operating systems before Windows Vista, a culture name that includes a script uses the pattern `Languagecode2 - country/regioncode2 - scripttag`, for example, `uz-UZ-Cyr1` for Uzbek (Cyrillic, Uzbekistan).

A neutral culture is specified by only the two-letter, lowercase language code. For example, `fr` specifies the neutral culture for French, and `de` specifies the neutral culture for German.

ⓘ Note

There are two culture names that contradict this rule. The cultures Chinese (Simplified), named `zh-Hans`, and Chinese (Traditional), named `zh-Hant`, are neutral cultures. The culture names represent the current standard and should be used unless you have a reason for using the older names `zh-CHS` and `zh-CHT`.

A culture identifier is a standard international numeric abbreviation and has the components necessary to uniquely identify one of the installed cultures. Your application can use predefined culture identifiers or define custom identifiers.

Certain predefined culture names and identifiers are used by this and other classes in the [System.Globalization](#) namespace. For detailed culture information for Windows systems, see the **Language tag** column in the [list of language/region names supported by Windows](#). Culture names follow the standard defined by [BCP 47](#).

The culture names and identifiers represent only a subset of cultures that can be found on a particular computer. Windows versions or service packs can change the available cultures. Applications can add custom cultures using the [CultureAndRegionInfoBuilder](#) class. Users can add their own custom cultures using the [Microsoft Locale Builder](#) tool. Microsoft Locale Builder is written in managed code using the [CultureAndRegionInfoBuilder](#) class.

Several distinct names are closely associated with a culture, notably the names associated with the following class members:

- [CultureInfo.ToString](#)
- [CultureInfo.Name](#)
- [CompareInfo.Name](#)

Invariant, neutral, and specific cultures

The cultures are generally grouped into three sets: invariant cultures, neutral cultures, and specific cultures.

An invariant culture is culture-insensitive. Your application specifies the invariant culture by name using an empty string ("") or by its identifier. [InvariantCulture](#) defines an instance of the invariant culture. It is associated with the English language but not with any country/region. It is used in almost any method in the [Globalization](#) namespace that requires a culture.

A neutral culture is a culture that is associated with a language but not with a country/region. A specific culture is a culture that is associated with a language and a country/region. For example, `fr` is the neutral name for the French culture, and `fr-FR` is the name of the specific French (France) culture. Note that Chinese (Simplified) and Chinese (Traditional) are also considered neutral cultures.

Creating an instance of a [CompareInfo](#) class for a neutral culture is not recommended because the data it contains is arbitrary. To display and sort data, specify both the language and region. Additionally, the [Name](#) property of a [CompareInfo](#) object created for a neutral culture returns only the country and does not include the region.

The defined cultures have a hierarchy in which the parent of a specific culture is a neutral culture and the parent of a neutral culture is the invariant culture. The [Parent](#) property contains the neutral culture associated with a specific culture. Custom cultures should define the [Parent](#) property in conformance with this pattern.

If the resources for a specific culture are not available in the operating system, the resources for the associated neutral culture are used. If the resources for the neutral culture are not available, the resources embedded in the main assembly are used. For more information on the resource fallback process, see [Packaging and Deploying Resources](#).

The list of locales in the Windows API is slightly different from the list of cultures supported by .NET. If interoperability with Windows is required, for example, through the p/invoke mechanism, the application should use a specific culture that's defined for the operating system. Use of the specific culture ensures consistency with the equivalent Windows locale, which is identified with a locale identifier that is the same as [LCID](#).

A [DateTimeFormatInfo](#) or a [NumberFormatInfo](#) can be created only for the invariant culture or for specific cultures, not for neutral cultures.

If [DateTimeFormatInfo.Calendar](#) is the [TaiwanCalendar](#) but the [Thread.CurrentCulture](#) is not set to `zh-TW`, then [DateTimeFormatInfo.NativeCalendarName](#),

`DateTimeFormatInfo.GetEraName`, and `DateTimeFormatInfo.GetAbbreviatedEraName` return an empty string ("").

Custom cultures

On Windows, you can create custom locales. For more information, see [Custom locales](#).

CultureInfo and cultural data

.NET derives its cultural data from one of a variety of sources, depending on implementation, platform, and version:

- In all versions of .NET (Core) running on Unix platforms or Windows 10 and later versions, cultural data is provided by the [International Components for Unicode \(ICU\) Library](#). The specific version of the ICU Library depends on the individual operating system.
- In all versions of .NET (Core) running on Windows 9 and earlier versions, cultural data is provided by the Windows operating system.
- In .NET Framework 4 and later versions, cultural data is provided by the Windows operating system.
- In .NET Framework 3.5 and earlier versions, cultural data is provided by both the Windows operating system and .NET Framework.

Because of this, a culture available on a particular .NET implementation, platform, or version may not be available on a different .NET implementation, platform, or version.

Some `CultureInfo` objects differ depending on the underlying platform. In particular, `zh-CN`, or Chinese (Simplified, China) and `zh-TW`, or Chinese (Traditional, Taiwan), are available cultures on Windows systems, but they are aliased cultures on Unix systems. "zh-CN" is an alias for the "zh-Hans-CN" culture, and "zh-TW" is an alias for the "zh-Hant-TW" culture. Aliased cultures are not returned by calls to the `GetCultures` method and may have different property values, including different `Parent` cultures, than their Windows counterparts. For the `zh-CN` and `zh-TW` cultures, these differences include the following:

- On Windows systems, the parent culture of the "zh-CN" culture is "zh-Hans", and the parent culture of the "zh-TW" culture is "zh-Hant". The parent culture of both these cultures is "zh". On Unix systems, the parents of both cultures are "zh". This means that, if you don't provide culture-specific resources for the "zh-CN" or "zh-TW" cultures but do provide resources for the neutral "zh-Hans" or "zh-Hant" culture, your application will load the resources for the neutral culture on Windows

but not on Unix. On Unix systems, you must explicitly set the thread's [CurrentUICulture](#) to either "zh-Hans" or "zh-Hant".

- On Windows systems, calling [CultureInfo.Equals](#) on an instance that represents the "zh-CN" culture and passing it a "zh-Hans-CN" instance returns `true`. On Unix systems, the method call returns `false`. This behavior also applies to calling [Equals](#) on a "zh-TW" [CultureInfo](#) instance and passing it a "zh-Hant-Tw" instance.

Dynamic culture data

Except for the invariant culture, culture data is dynamic. This is true even for the predefined cultures. For example, countries or regions adopt new currencies, change their spellings of words, or change their preferred calendar, and culture definitions change to track this. Custom cultures are subject to change without notice, and any specific culture might be overridden by a custom replacement culture. Also, as discussed below, an individual user can override cultural preferences. Applications should always obtain culture data at run time.

⊗ Caution

When saving data, your application should use the invariant culture, a binary format, or a specific culture-independent format. Data saved according to the current values associated with a particular culture, other than the invariant culture, might become unreadable or might change in meaning if that culture changes.

The current culture and current UI culture

Every thread in a .NET application has a current culture and a current UI culture. The current culture determines the formatting conventions for dates, times, numbers, and currency values, the sort order of text, casing conventions, and the ways in which strings are compared. The current UI culture is used to retrieve culture-specific resources at run time.

ⓘ Note

For information on how the current and current UI culture is determined on a per-thread basis, see the [Culture and threads](#) section. For information on how the current and current UI culture is determined on threads executing in a new application domain, and on threads that cross application domain boundaries, see the [Culture and application domains](#) section. For information on how the current

and current UI culture is determined on threads performing task-based asynchronous operations, see the [Culture and task-based asynchronous operations](#) section.

For more detailed information on the current culture, see the [CultureInfo.CurrentCulture](#) property. For more detailed information on the current UI culture, see the [CultureInfo.CurrentUICulture](#) property topic.

Retrieve the current and current UI cultures

You can get a [CultureInfo](#) object that represents the current culture in either of two ways:

- By retrieving the value of the [CultureInfo.CurrentCulture](#) property.
- By retrieving the value of the [Thread.CurrentThread.CurrentCulture](#) property.

The following example retrieves both property values, compares them to show that they are equal, and displays the name of the current culture.

C#

```
using System;
using System.Globalization;
using System.Threading;

public class CurrentCultureEx
{
    public static void Main()
    {
        CultureInfo culture1 = CultureInfo.CurrentCulture;
        CultureInfo culture2 = Thread.CurrentThread.CurrentCulture;
        Console.WriteLine("The current culture is {0}", culture1.Name);
        Console.WriteLine("The two CultureInfo objects are equal: {0}",
                          culture1 == culture2);
    }
}

// The example displays output like the following:
//      The current culture is en-US
//      The two CultureInfo objects are equal: True
```

You can get a [CultureInfo](#) object that represents the current UI culture in either of two ways:

- By retrieving the value of the [CultureInfo.CurrentUICulture](#) property.
- By retrieving the value of the [Thread.CurrentThread.CurrentUICulture](#) property.

The following example retrieves both property values, compares them to show that they are equal, and displays the name of the current UI culture.

```
C#  
  
using System;  
using System.Globalization;  
using System.Threading;  
  
public class CurrentUIEx  
{  
    public static void Main()  
    {  
        CultureInfo uiCulture1 = CultureInfo.CurrentCulture;  
        CultureInfo uiCulture2 = Thread.CurrentThread.CurrentCulture;  
        Console.WriteLine("The current UI culture is {0}", uiCulture1.Name);  
        Console.WriteLine("The two CultureInfo objects are equal: {0}",  
                          uiCulture1 == uiCulture2);  
    }  
}  
// The example displays output like the following:  
//     The current UI culture is en-US  
//     The two CultureInfo objects are equal: True
```

Set the current and current UI cultures

To change the culture and UI culture of a thread, do the following:

1. Instantiate a [CultureInfo](#) object that represents that culture by calling a [CultureInfo](#) class constructor and passing it the name of the culture. The [CultureInfo\(String\)](#) constructor instantiates a [CultureInfo](#) object that reflects user overrides if the new culture is the same as the current Windows culture. The [CultureInfo\(String, Boolean\)](#) constructor allows you to specify whether the newly instantiated [CultureInfo](#) object reflects user overrides if the new culture is the same as the current Windows culture.
2. Assign the [CultureInfo](#) object to the [CultureInfo.CurrentCulture](#) or [CultureInfo.CurrentUICulture](#) property on .NET Core and .NET Framework 4.6 and later versions. (On .NET Framework 4.5.2 and earlier versions, you can assign the [CultureInfo](#) object to the [Thread.CurrentCulture](#) or [Thread.CurrentUICulture](#) property.)

The following example retrieves the current culture. If it is anything other than the French (France) culture, it changes the current culture to French (France). Otherwise, it changes the current culture to French (Luxembourg).

C#

```
using System;
using System.Globalization;

public class ChangeEx1
{
    public static void Main()
    {
        CultureInfo current = CultureInfo.CurrentCulture;
        Console.WriteLine("The current culture is {0}", current.Name);
        CultureInfo newCulture;
        if (current.Name.Equals("fr-FR"))
            newCulture = new CultureInfo("fr-LU");
        else
            newCulture = new CultureInfo("fr-FR");

        CultureInfo.CurrentCulture = newCulture;
        Console.WriteLine("The current culture is now {0}",
                          CultureInfo.CurrentCulture.Name);
    }
}

// The example displays output like the following:
//      The current culture is en-US
//      The current culture is now fr-FR
```

The following example retrieves the current culture. If it is anything other the Slovenian (Slovenia) culture, it changes the current culture to Slovenian (Slovenia). Otherwise, it changes the current culture to Croatian (Croatia).

C#

```
using System;
using System.Globalization;

public class ChangeUICultureEx
{
    public static void Main()
    {
        CultureInfo current = CultureInfo.CurrentCulture;
        Console.WriteLine("The current UI culture is {0}", current.Name);
        CultureInfo newUICulture;
        if (current.Name.Equals("sl-SI"))
            newUICulture = new CultureInfo("hr-HR");
        else
            newUICulture = new CultureInfo("sl-SI");

        CultureInfo.CurrentCulture = newUICulture;
        Console.WriteLine("The current UI culture is now {0}",
                          CultureInfo.CurrentCulture.Name);
    }
}
```

```
// The example displays output like the following:  
//      The current UI culture is en-US  
//      The current UI culture is now sl-SI
```

Get all cultures

You can retrieve an array of specific categories of cultures or of all the cultures available on the local computer by calling the [GetCultures](#) method. For example, you can retrieve custom cultures, specific cultures, or neutral cultures either alone or in combination.

The following example calls the [GetCultures](#) method twice, first with the [System.Globalization.CultureTypes](#) enumeration member to retrieve all custom cultures, and then with the [System.Globalization.CultureTypes](#) enumeration member to retrieve all replacement cultures.

C#

```
using System;  
using System.Globalization;  
  
public class GetCulturesEx  
{  
    public static void Main()  
    {  
        // Get all custom cultures.  
        CultureInfo[] custom =  
CultureInfo.GetCultures(CultureTypes.UserCustomCulture);  
        if (custom.Length == 0)  
        {  
            Console.WriteLine("There are no user-defined custom cultures.");  
        }  
        else  
        {  
            Console.WriteLine("Custom cultures:");  
            foreach (var culture in custom)  
                Console.WriteLine("    {0} -- {1}", culture.Name,  
culture.DisplayName);  
        }  
        Console.WriteLine();  
  
        // Get all replacement cultures.  
        CultureInfo[] replacements =  
CultureInfo.GetCultures(CultureTypes.ReplacementCultures);  
        if (replacements.Length == 0)  
        {  
            Console.WriteLine("There are no replacement cultures.");  
        }  
        else  
        {  
            Console.WriteLine("Replacement cultures:");  
        }  
    }  
}
```

```

        foreach (var culture in replacements)
            Console.WriteLine("  {0} -- {1}", culture.Name,
culture.DisplayName);
    }
    Console.WriteLine();
}
// The example displays output like the following:
//   Custom cultures:
//     x-en-US-sample -- English (United States)
//     fj-FJ -- Boumaa Fijian (Viti)
//
// There are no replacement cultures.

```

Culture and threads

When a new application thread is started, its current culture and current UI culture are defined by the current system culture, and not by the current thread culture. The following example illustrates the difference. It sets the current culture and current UI culture of an application thread to the French (France) culture (fr-FR). If the current culture is already fr-FR, the example sets it to the English (United States) culture (en-US). It displays three random numbers as currency values and then creates a new thread, which, in turn, displays three more random numbers as currency values. But as the output from the example shows, the currency values displayed by the new thread do not reflect the formatting conventions of the French (France) culture, unlike the output from the main application thread.

C#

```

using System;
using System.Globalization;
using System.Threading;

public class DefaultThreadEx
{
    static Random rnd = new Random();

    public static void Main()
    {
        if (Thread.CurrentThread.CurrentCulture.Name != "fr-FR")
        {
            // If current culture is not fr-FR, set culture to fr-FR.
            Thread.CurrentThread.CurrentCulture =
CultureInfo.CreateSpecificCulture("fr-FR");
            Thread.CurrentThread.CurrentUICulture =
CultureInfo.CreateSpecificCulture("fr-FR");
        }
        else

```

```

    {
        // Set culture to en-US.
        Thread.CurrentThread.CurrentCulture =
CultureInfo.CreateSpecificCulture("en-US");
        Thread.CurrentThread.CurrentUICulture =
CultureInfo.CreateSpecificCulture("en-US");
    }
    ThreadProc();

    Thread worker = new Thread(ThreadProc);
    worker.Name = "WorkerThread";
    worker.Start();
}

private static void DisplayThreadInfo()
{
    Console.WriteLine("\nCurrent Thread Name: '{0}'",
                      Thread.CurrentThread.Name);
    Console.WriteLine("Current Thread Culture/UI Culture: {0}/{1}",
                      Thread.CurrentThread.CurrentCulture.Name,
                      Thread.CurrentThread.CurrentUICulture.Name);
}

private static void DisplayValues()
{
    // Create new thread and display three random numbers.
    Console.WriteLine("Some currency values:");
    for (int ctr = 0; ctr <= 3; ctr++)
        Console.WriteLine("    {0:C2}", rnd.NextDouble() * 10);
}

private static void ThreadProc()
{
    DisplayThreadInfo();
    DisplayValues();
}
}

// The example displays output similar to the following:
//      Current Thread Name: ''
//      Current Thread Culture/UI Culture: fr-FR/fr-FR
//      Some currency values:
//          8,11 €
//          1,48 €
//          8,99 €
//          9,04 €
//
//      Current Thread Name: 'WorkerThread'
//      Current Thread Culture/UI Culture: en-US/en-US
//      Some currency values:
//          $6.72
//          $6.35
//          $2.90
//          $7.72

```

You can set the culture and UI culture of all threads in an application domain by assigning a [CultureInfo](#) object that represents that culture to the [DefaultThreadCurrentCulture](#) and [DefaultThreadCurrentUICulture](#) properties. The following example uses these properties to ensure that all threads in the default application domain share the same culture.

C#

```
using System;
using System.Globalization;
using System.Threading;

public class SetThreadsEx
{
    static Random rnd = new Random();

    public static void Main()
    {
        if (Thread.CurrentThread.CurrentCulture.Name != "fr-FR")
        {
            // If current culture is not fr-FR, set culture to fr-FR.
            CultureInfo.DefaultThreadCurrentCulture =
CultureInfo.CreateSpecificCulture("fr-FR");
            CultureInfo.DefaultThreadCurrentUICulture =
CultureInfo.CreateSpecificCulture("fr-FR");
        }
        else
        {
            // Set culture to en-US.
            CultureInfo.DefaultThreadCurrentCulture =
CultureInfo.CreateSpecificCulture("en-US");
            CultureInfo.DefaultThreadCurrentUICulture =
CultureInfo.CreateSpecificCulture("en-US");
        }
        ThreadProc();
    }

    Thread worker = new Thread(SetThreadsEx.ThreadProc);
    worker.Name = "WorkerThread";
    worker.Start();
}

private static void DisplayThreadInfo()
{
    Console.WriteLine("\nCurrent Thread Name: '{0}'",
                    Thread.CurrentThread.Name);
    Console.WriteLine("Current Thread Culture/UI Culture: {0}/{1}",
                    Thread.CurrentThread.CurrentCulture.Name,
                    Thread.CurrentThread.CurrentUICulture.Name);
}

private static void DisplayValues()
{
    // Create new thread and display three random numbers.
}
```

```

        Console.WriteLine("Some currency values:");
        for (int ctr = 0; ctr <= 3; ctr++)
            Console.WriteLine("    {0:C2}", rnd.NextDouble() * 10);
    }

    private static void ThreadProc()
    {
        DisplayThreadInfo();
        DisplayValues();
    }
}

// The example displays output similar to the following:
//      Current Thread Name: ''
//      Current Thread Culture/UI Culture: fr-FR/fr-FR
//      Some currency values:
//          6,83 €
//          3,47 €
//          6,07 €
//          1,70 €
//
//      Current Thread Name: 'WorkerThread'
//      Current Thread Culture/UI Culture: fr-FR/fr-FR
//      Some currency values:
//          9,54 €
//          9,50 €
//          0,58 €
//          6,91 €

```

Warning

Although the [DefaultThreadCurrentCulture](#) and [DefaultThreadCurrentUICulture](#) properties are static members, they define the default culture and default UI culture only for the application domain that is current at the time these property values are set. For more information, see the next section, [Culture and application domains](#).

When you assign values to the [DefaultThreadCurrentCulture](#) and [DefaultThreadCurrentUICulture](#) properties, the culture and UI culture of the threads in the application domain also change if they have not explicitly been assigned a culture. However, these threads reflect the new culture settings only while they execute in the current application domain. If these threads execute in another application domain, their culture becomes the default culture defined for that application domain. As a result, we recommend that you always set the culture of the main application thread, and not rely on the [DefaultThreadCurrentCulture](#) and [DefaultThreadCurrentUICulture](#) properties to change it.

Culture and application domains

`DefaultThreadCurrentCulture` and `DefaultThreadCurrentUICulture` are static properties that explicitly define a default culture only for the application domain that is current when the property value is set or retrieved. The following example sets the default culture and default UI culture in the default application domain to French (France), and then uses the `AppDomainSetup` class and the `AppDomainInitializer` delegate to set the default culture and UI culture in a new application domain to Russian (Russia). A single thread then executes two methods in each application domain. Note that the thread's culture and UI culture are not explicitly set; they are derived from the default culture and UI culture of the application domain in which the thread is executing. Note also that the `DefaultThreadCurrentCulture` and `DefaultThreadCurrentUICulture` properties return the default `CultureInfo` values of the application domain that is current when the method call is made.

C#

```
using System;
using System.Globalization;

public class Example
{
    public static void Main()
    {
        // Set the default culture and display the current date in the
        current application domain.
        Info info1 = new Info();
        SetAppDomainCultures("fr-FR");

        // Create a second application domain.
        AppDomainSetup setup = new AppDomainSetup();
        setup.AppDomainInitializer = SetAppDomainCultures;
        setup.AppDomainInitializerArguments = new string[] { "ru-RU" };
        AppDomain domain = AppDomain.CreateDomain("Domain2", null, setup);
        // Create an Info object in the new application domain.
        Info info2 =
(Info)domain.CreateInstanceAndUnwrap(typeof(Example).Assembly.FullName,
                                         "Info");

        // Execute methods in the two application domains.
        info2.DisplayDate();
        info2.DisplayCultures();

        info1.DisplayDate();
        info1.DisplayCultures();
    }

    public static void SetAppDomainCultures(string[] names)
    {
        SetAppDomainCultures(names[0]);
    }
}
```

```

public static void SetAppDomainCultures(string name)
{
    try
    {
        CultureInfo.DefaultThreadCurrentCulture =
CultureInfo.CreateSpecificCulture(name);
        CultureInfo.DefaultThreadCurrentUICulture =
CultureInfo.CreateSpecificCulture(name);
    }
    // If an exception occurs, we'll just fall back to the system
default.
    catch (CultureNotFoundException)
    {
        return;
    }
    catch (ArgumentException)
    {
        return;
    }
}
}

public class Info : MarshalByRefObject
{
    public void DisplayDate()
    {
        Console.WriteLine("Today is {0:D}", DateTime.Now);
    }

    public void DisplayCultures()
    {
        Console.WriteLine("Application domain is {0}",
AppDomain.CurrentDomain.Id);
        Console.WriteLine("Default Culture: {0}",
CultureInfo.DefaultThreadCurrentCulture);
        Console.WriteLine("Default UI Culture: {0}",
CultureInfo.DefaultThreadCurrentUICulture);
    }
}
// The example displays the following output:
//      Today is 14 октября 2011 г.
//      Application domain is 2
//      Default Culture: ru-RU
//      Default UI Culture: ru-RU
//      Today is vendredi 14 octobre 2011
//      Application domain is 1
//      Default Culture: fr-FR
//      Default UI Culture: fr-FR

```

For more information about cultures and application domains, see the "Application Domains and Threads" section in the [Application Domains](#) topic.

Culture and task-based asynchronous operations

The [task-based asynchronous programming pattern](#) uses `Task` and `Task<TResult>` objects to asynchronously execute delegates on thread pool threads. The specific thread on which a particular task runs is not known in advance, but is determined only at runtime.

For apps that target .NET Framework 4.6 or a later version, culture is part of an asynchronous operation's context. In other words, asynchronous operations by default inherit the values of the `CurrentCulture` and `CurrentUICulture` properties of the thread from which they are launched. If the current culture or current UI culture differs from the system culture, the current culture crosses thread boundaries and becomes the current culture of the thread pool thread that is executing an asynchronous operation.

The following example provides a simple illustration. The example defines a `Func<TResult>` delegate, `formatDelegate`, that returns some numbers formatted as currency values. The example changes the current system culture to either French (France) or, if French (France) is already the current culture, English (United States). It then:

- Invokes the delegate directly so that it runs synchronously on the main app thread.
- Creates a task that executes the delegate asynchronously on a thread pool thread.
- Creates a task that executes the delegate synchronously on the main app thread by calling the `Task.RunSynchronously` method.

As the output from the example shows, when the current culture is changed to French (France), the current culture of the thread from which tasks are invoked asynchronously becomes the current culture for that asynchronous operation.

C#

```
using System;
using System.Globalization;
using System.Threading;
using System.Threading.Tasks;

public class AsyncCultureEx1
{
    public static void Main()
    {
        decimal[] values = { 163025412.32m, 18905365.59m };
        string formatString = "C2";

        string FormatDelegate()
```

```

    {
        string output = $"Formatting using the
{CultureInfo.CurrentCulture.Name} " +
            "culture on thread {Thread.CurrentThread.ManagedThreadId}.\n";
        foreach (decimal value in values)
            output += $"{value.ToString(formatString)}    ";

        output += Environment.NewLine;
        return output;
    }

    Console.WriteLine($"The example is running on thread
{Thread.CurrentThread.ManagedThreadId}");
    // Make the current culture different from the system culture.
    Console.WriteLine($"The current culture is
{CultureInfo.CurrentCulture.Name}");
    if (CultureInfo.CurrentCulture.Name == "fr-FR")
        Thread.CurrentThread.CurrentCulture = new CultureInfo("en-US");
    else
        Thread.CurrentThread.CurrentCulture = new CultureInfo("fr-FR");

    Console.WriteLine($"Changed the current culture to
{CultureInfo.CurrentCulture.Name}.\n");

    // Execute the delegate synchronously.
    Console.WriteLine("Executing the delegate synchronously:");
    Console.WriteLine(FormatDelegate());

    // Call an async delegate to format the values using one format
string.
    Console.WriteLine("Executing a task asynchronously:");
    var t1 = Task.Run(FormatDelegate);
    Console.WriteLine(t1.Result);

    Console.WriteLine("Executing a task synchronously:");
    var t2 = new Task<string>(FormatDelegate);
    t2.RunSynchronously();
    Console.WriteLine(t2.Result);
}
}

// The example displays the following output:
//      The example is running on thread 1
//      The current culture is en-US
//      Changed the current culture to fr-FR.

//
//      Executing the delegate synchronously:
//      Formatting using the fr-FR culture on thread 1.
//      163 025 412,32 €  18 905 365,59 €

//
//      Executing a task asynchronously:
//      Formatting using the fr-FR culture on thread 3.
//      163 025 412,32 €  18 905 365,59 €

//
//      Executing a task synchronously:

```

```
//      Formatting using the fr-FR culture on thread 1.  
//      163 025 412,32 €    18 905 365,59 €
```

For apps that target versions of .NET Framework from .NET Framework 4.5 and later but prior to .NET Framework 4.6, you can use the [DefaultThreadCurrentCulture](#) and [DefaultThreadCurrentUICulture](#) properties to ensure that the culture of the calling thread is used in asynchronous tasks that execute on thread pool threads. The following example is identical to the previous example, except that it uses the [DefaultThreadCurrentCulture](#) property to ensure that thread pool threads have the same culture as the main app thread.

C#

```
using System;  
using System.Globalization;  
using System.Threading;  
using System.Threading.Tasks;  
  
public class AsyncCultureEx3  
{  
    public static void Main()  
    {  
        decimal[] values = { 163025412.32m, 18905365.59m };  
        string formatString = "C2";  
        Func<String> formatDelegate = () =>  
        {  
            string output = String.Format("Formatting using the {0} culture  
on thread {1}.\n",  
                CultureInfo.CurrentCulture.Name,  
                Thread.CurrentThread.ManagedThreadId);  
            foreach (var value in values)  
                output += String.Format("{0}    ",  
                    value.ToString(formatString));  
  
            output += Environment.NewLine;  
            return output;  
        };  
  
        Console.WriteLine("The example is running on thread {0}",  
            Thread.CurrentThread.ManagedThreadId);  
        // Make the current culture different from the system culture.  
        Console.WriteLine("The current culture is {0}",  
            CultureInfo.CurrentCulture.Name);  
        if (CultureInfo.CurrentCulture.Name == "fr-FR")  
            Thread.CurrentThread.CurrentCulture = new CultureInfo("en-US");  
        else  
            Thread.CurrentThread.CurrentCulture = new CultureInfo("fr-FR");  
  
        Console.WriteLine("Changed the current culture to {0}.\n",
```

```

        CultureInfo.CurrentCulture.Name);
CultureInfo.DefaultThreadCurrentCulture =
CultureInfo.CurrentCulture;

    // Execute the delegate synchronously.
Console.WriteLine("Executing the delegate synchronously:");
Console.WriteLine(formatDelegate());

    // Call an async delegate to format the values using one format
string.
Console.WriteLine("Executing a task asynchronously:");
var t1 = Task.Run(formatDelegate);
Console.WriteLine(t1.Result);

Console.WriteLine("Executing a task synchronously:");
var t2 = new Task<String>(formatDelegate);
t2.RunSynchronously();
Console.WriteLine(t2.Result);
}

}

// The example displays the following output:
//   The example is running on thread 1
//   The current culture is en-US
//   Changed the current culture to fr-FR.
//
//   Executing the delegate synchronously:
//   Formatting using the fr-FR culture on thread 1.
//   163 025 412,32 €   18 905 365,59 €

//
//   Executing a task asynchronously:
//   Formatting using the fr-FR culture on thread 3.
//   163 025 412,32 €   18 905 365,59 €

//
//   Executing a task synchronously:
//   Formatting using the fr-FR culture on thread 1.
//   163 025 412,32 €   18 905 365,59 €

```

`DefaultThreadCurrentCulture` and `DefaultThreadCurrentUICulture` are per-app domain properties; that is, they establish a default culture for all threads not explicitly assigned a culture in a specific application domain. However, for apps that target .NET Framework 4.6 or later, the culture of the calling thread remains part of an asynchronous task's context even if the task crosses app domain boundaries.

CultureInfo object serialization

When a `CultureInfo` object is serialized, all that is actually stored is `Name` and `UseUserOverride`. It is successfully deserialized only in an environment where that `Name` has the same meaning. The following three examples show why this is not always the case:

- If the `CultureTypes` property value is `CultureTypes.InstalledWin32Cultures`, and if that culture was first introduced in a particular version of the Windows operating system, it is not possible to deserialize it on an earlier version of Windows. For example, if a culture was introduced in Windows 10, it cannot be deserialized on Windows 8.
- If the `CultureTypes` value is `CultureTypes.UserCustomCulture`, and the computer on which it is serialized does not have this user custom culture installed, it is not possible to deserialize it.
- If the `CultureTypes` value is `CultureTypes.ReplacementCultures`, and the computer on which it is serialized does not have this replacement culture, it serializes to the same name, but not all of the same characteristics. For example, if `en-US` is a replacement culture on computer A, but not on computer B, and if a `CultureInfo` object referring to this culture is serialized on computer A and serialized on computer B, then none of the custom characteristics of the culture are transmitted. The culture serializes successfully, but with a different meaning.

Control Panel overrides

The user might choose to override some of the values associated with the current culture of Windows through the regional and language options portion of Control Panel. For example, the user might choose to display the date in a different format or to use a currency other than the default for the culture. In general, your applications should honor these user overrides.

If `UseUserOverride` is `true` and the specified culture matches the current culture of Windows, the `CultureInfo` uses those overrides, including user settings for the properties of the `DateTimeFormatInfo` instance returned by the `DateTimeFormat` property, and the properties of the `NumberFormatInfo` instance returned by the `NumberFormat` property. If the user settings are incompatible with the culture associated with the `CultureInfo`, for example, if the selected calendar is not one of the `OptionalCalendars`, the results of the methods and the values of the properties are undefined.

Alternate sort orders

Some cultures support more than one sort order. For example:

- The Spanish (Spain) culture has two sort orders: the default international sort order, and the traditional sort order. When you instantiate a `CultureInfo` object with the `es-ES` culture name, the international sort order is used. When you

instantiate a [CultureInfo](#) object with the `es-ES-tradnl` culture name, the traditional sort order is used.

- The `zh-CN` (Chinese (Simplified, PRC)) culture supports two sort orders: by pronunciation (the default) and by stroke count. When you instantiate a [CultureInfo](#) object with the `zh-CN` culture name, the default sort order is used. When you instantiate a [CultureInfo](#) object with a local identifier of `0x00020804`, strings are sorted by stroke count.

The following table lists the cultures that support alternate sort orders and the identifiers for the default and alternate sort orders.

[\[+\] Expand table](#)

| Culture name | Culture | Default sort name and identifier | Alternate sort name and identifier |
|--------------|-------------------------|---|---|
| es-ES | Spanish (Spain) | International: <code>0x00000C0A</code> | Traditional: <code>0x0000040A</code> |
| zh-TW | Chinese (Taiwan) | Stroke Count: <code>0x00000404</code> | Bopomofo: <code>0x00030404</code> |
| zh-CN | Chinese (PRC) | Pronunciation: <code>0x00000804</code> | Stroke Count: <code>0x00020804</code> |
| zh-HK | Chinese (Hong Kong SAR) | Stroke Count: <code>0x00000c04</code> | Stroke Count: <code>0x00020c04</code> |
| zh-SG | Chinese (Singapore) | Pronunciation: <code>0x00001004</code> | Stroke Count: <code>0x00021004</code> |
| zh-MO | Chinese (Macao SAR) | Pronunciation: <code>0x00001404</code> | Stroke Count: <code>0x00021404</code> |
| ja-JP | Japanese (Japan) | Default: <code>0x00000411</code> | Unicode: <code>0x00010411</code> |
| ko-KR | Korean (Korea) | Default: <code>0x00000412</code> | Korean Xwansung - Unicode: <code>0x00010412</code> |
| de-DE | German (Germany) | Dictionary: <code>0x00000407</code> | Phone Book Sort DIN: <code>0x00010407</code> |
| hu-HU | Hungarian (Hungary) | Default: <code>0x0000040e</code> | Technical Sort: <code>0x0001040e</code> |
| ka-GE | Georgian (Georgia) | Traditional: <code>0x00000437</code> | Modern Sort: <code>0x00010437</code> |

The current culture and UWP apps

In Universal Windows Platform (UWP) apps, the [CurrentCulture](#) and [CurrentUICulture](#) properties are read-write, just as they are in .NET Framework and .NET Core apps. However, UWP apps recognize a single culture. The [CurrentCulture](#) and [CurrentUICulture](#) properties map to the first value in the [Windows.ApplicationModel.Resources.Core.ResourceManager.DefaultContext.Languages](#) collection.

In .NET apps, the current culture is a per-thread setting, and the [CurrentCulture](#) and [CurrentUICulture](#) properties reflect the culture and UI culture of the current thread only.

In UWP apps, the current culture maps to the [Windows.ApplicationModel.Resources.Core.ResourceManager.DefaultContext.Languages](#) collection, which is a global setting. Setting the [CurrentCulture](#) or [CurrentUICulture](#) property changes the culture of the entire app; culture cannot be set on a per-thread basis.

 [Collaborate with us on GitHub](#)

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



.NET feedback

.NET is an open source project.
Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

DateTimeFormatInfo class

Article • 12/30/2023

This article provides supplementary remarks to the reference documentation for this API.

The properties of the [DateTimeFormatInfo](#) class contain culture-specific information for formatting or parsing date and time values such as the following:

- The patterns used to format date values.
- The patterns used to format time values.
- The names of the days of the week.
- The names of the months of the year.
- The A.M. and P.M. designators used in time values.
- The calendar in which dates are expressed.

Instantiate a DateTimeFormatInfo object

A [DateTimeFormatInfo](#) object can represent the formatting conventions of the invariant culture, a specific culture, a neutral culture, or the current culture. This section discusses how to instantiate each type of [DateTimeFormatInfo](#) object.

Instantiate a DateTimeFormatInfo object for the invariant culture

The invariant culture represents a culture that is culture-insensitive. It is based on the English language, but not on any specific English-speaking country/region. Although the data of specific cultures can be dynamic and can change to reflect new cultural conventions or user preferences, the data of the invariant culture does not change. You can instantiate a [DateTimeFormatInfo](#) object that represents the formatting conventions of the invariant culture in the following ways:

- By retrieving the value of the [InvariantInfo](#) property. The returned [DateTimeFormatInfo](#) object is read-only.
- By calling the parameterless [DateTimeFormatInfo](#) constructor. The returned [DateTimeFormatInfo](#) object is read/write.
- By retrieving the value of the [DateTimeFormat](#) property from the [CultureInfo](#) object that is returned by the [CultureInfo.InvariantCulture](#) property. The returned [DateTimeFormatInfo](#) object is read-only.

The following example uses each of these methods to instantiate a [DateTimeFormatInfo](#) object that represents the invariant culture. It then indicates whether the object is read-only.

C#

```
System.Globalization.DateTimeFormatInfo dtfi;

dtfi = System.Globalization.DateTimeFormatInfo.InvariantInfo;
Console.WriteLine(dtfi.IsReadOnly);

dtfi = new System.Globalization.DateTimeFormatInfo();
Console.WriteLine(dtfi.IsReadOnly);

dtfi = System.Globalization.CultureInfo.InvariantCulture.DateTimeFormat;
Console.WriteLine(dtfi.IsReadOnly);
// The example displays the following output:
//      True
//      False
//      True
```

Instantiate a [DateTimeFormatInfo](#) object for a specific culture

A specific culture represents a language that is spoken in a particular country/region. For example, en-US is a specific culture that represents the English language spoken in the United States, and en-CA is a specific culture that represents the English language spoken in Canada. You can instantiate a [DateTimeFormatInfo](#) object that represents the formatting conventions of a specific culture in the following ways:

- By calling the [CultureInfo.GetCultureInfo\(String\)](#) method and retrieving the value of the returned [CultureInfo](#) object's [CultureInfo.DateTimeFormat](#) property. The returned [DateTimeFormatInfo](#) object is read-only.
- By passing the static [GetInstance](#) method a [CultureInfo](#) object that represents the culture whose [DateTimeFormatInfo](#) object you want to retrieve. The returned [DateTimeFormatInfo](#) object is read/write.
- By calling the static [CultureInfo.CreateSpecificCulture](#) method and retrieving the value of the returned [CultureInfo](#) object's [CultureInfo.DateTimeFormat](#) property. The returned [DateTimeFormatInfo](#) object is read/write.
- By calling the [CultureInfo.CultureInfo](#) class constructor and retrieving the value of the returned [CultureInfo](#) object's [CultureInfo.DateTimeFormat](#) property. The returned [DateTimeFormatInfo](#) object is read/write.

The following example illustrates each of these ways to instantiate a [DateTimeFormatInfo](#) object and indicates whether the resulting object is read-only.

C#

```
System.Globalization.CultureInfo ci = null;
System.Globalization.DateTimeFormatInfo dtfi = null;

// Instantiate a culture using CreateSpecificCulture.
ci = System.Globalization.CultureInfo.CreateSpecificCulture("en-US");
dtfi = ci.DateTimeFormat;
Console.WriteLine("{0} from CreateSpecificCulture: {1}", ci.Name,
dtfi.IsReadOnly);

// Instantiate a culture using the CultureInfo constructor.
ci = new System.Globalization.CultureInfo("en-CA");
dtfi = ci.DateTimeFormat;
Console.WriteLine("{0} from CultureInfo constructor: {1}", ci.Name,
dtfi.IsReadOnly);

// Retrieve a culture by calling the GetCultureInfo method.
ci = System.Globalization.CultureInfo.GetCultureInfo("en-AU");
dtfi = ci.DateTimeFormat;
Console.WriteLine("{0} from GetCultureInfo: {1}", ci.Name, dtfi.IsReadOnly);

// Instantiate a DateTimeFormatInfo object by calling
DateTimeFormatInfo.GetInstance.
ci = System.Globalization.CultureInfo.CreateSpecificCulture("en-GB");
dtfi = System.Globalization.DateTimeFormatInfo.GetInstance(ci);
Console.WriteLine("{0} from GetInstance: {1}", ci.Name, dtfi.IsReadOnly);

// The example displays the following output:
//      en-US from CreateSpecificCulture: False
//      en-CA from CultureInfo constructor: False
//      en-AU from GetCultureInfo: True
//      en-GB from GetInstance: False
```

Instantiate a [DateTimeFormatInfo](#) object for a neutral culture

A neutral culture represents a culture or language that is independent of a country/region; it is typically the parent of one or more specific cultures. For example, Fr is a neutral culture for the French language and the parent of the fr-FR culture. You can instantiate a [DateTimeFormatInfo](#) object that represents the formatting conventions of a neutral culture in the same ways that you create a [DateTimeFormatInfo](#) object that represents the formatting conventions of a specific culture. In addition, you can retrieve a neutral culture's [DateTimeFormatInfo](#) object by retrieving a neutral culture from a specific culture's [CultureInfo.Parent](#) property and retrieving the [DateTimeFormatInfo](#)

object returned by its [CultureInfo.DateTimeFormat](#) property. Unless the parent culture represents the invariant culture, the returned [DateTimeFormatInfo](#) object is read/write. The following example illustrates these ways of instantiating a [DateTimeFormatInfo](#) object that represents a neutral culture.

C#

```
System.Globalization.CultureInfo specific, neutral;
System.Globalization.DateTimeFormatInfo dtfi;

// Instantiate a culture by creating a specific culture and using its Parent
// property.
specific = System.Globalization.CultureInfo.GetCultureInfo("fr-FR");
neutral = specific.Parent;
dtfi = neutral.DateTimeFormat;
Console.WriteLine("{0} from Parent property: {1}", neutral.Name,
dtfi.IsReadOnly);

dtfi = System.Globalization.CultureInfo.GetCultureInfo("fr-
FR").Parent.DateTimeFormat;
Console.WriteLine("{0} from Parent property: {1}", neutral.Name,
dtfi.IsReadOnly);

// Instantiate a neutral culture using the CultureInfo constructor.
neutral = new System.Globalization.CultureInfo("fr");
dtfi = neutral.DateTimeFormat;
Console.WriteLine("{0} from CultureInfo constructor: {1}", neutral.Name,
dtfi.IsReadOnly);

// Instantiate a culture using CreateSpecificCulture.
neutral = System.Globalization.CultureInfo.CreateSpecificCulture("fr");
dtfi = neutral.DateTimeFormat;
Console.WriteLine("{0} from CreateSpecificCulture: {1}", neutral.Name,
dtfi.IsReadOnly);

// Retrieve a culture by calling the GetCultureInfo method.
neutral = System.Globalization.CultureInfo.GetCultureInfo("fr");
dtfi = neutral.DateTimeFormat;
Console.WriteLine("{0} from GetCultureInfo: {1}", neutral.Name,
dtfi.IsReadOnly);

// Instantiate a DateTimeFormatInfo object by calling GetInstance.
neutral = System.Globalization.CultureInfo.CreateSpecificCulture("fr");
dtfi = System.Globalization.DateTimeFormatInfo.GetInstance(neutral);
Console.WriteLine("{0} from GetInstance: {1}", neutral.Name,
dtfi.IsReadOnly);

// The example displays the following output:
//      fr from Parent property: False
//      fr from Parent property: False
//      fr from CultureInfo constructor: False
//      fr-FR from CreateSpecificCulture: False
```

```
//      fr from GetCultureInfo: True
//      fr-FR from GetInstance: False
```

However, a neutral culture lacks culture-specific formatting information, because it is independent of a specific country/region. Instead of populating the [DateTimeFormatInfo](#) object with generic values, .NET returns a [DateTimeFormatInfo](#) object that reflects the formatting conventions of a specific culture that is a child of the neutral culture. For example, the [DateTimeFormatInfo](#) object for the neutral en culture reflects the formatting conventions of the en-US culture, and the [DateTimeFormatInfo](#) object for the fr culture reflects the formatting conventions of the fr-FR culture.

You can use code like the following to determine which specific culture's formatting conventions a neutral culture represents. The example uses reflection to compare the [DateTimeFormatInfo](#) properties of a neutral culture with the properties of a specific child culture. It considers two calendars to be equivalent if they are the same calendar type and, for Gregorian calendars, if their [GregorianCalendar.CalendarType](#) properties have identical values.

C#

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Globalization;
using System.Reflection;

public class InstantiateEx6
{
    public static void Main()
    {
        // Get all the neutral cultures
        List<String> names = new List<String>();
        Array.ForEach(CultureInfo.GetCultures(CultureTypes.NeutralCultures),
                     culture => names.Add(culture.Name));
        names.Sort();
        foreach (var name in names)
        {
            // Ignore the invariant culture.
            if (name == "") continue;

            ListSimilarChildCultures(name);
        }
    }

    private static void ListSimilarChildCultures(String name)
    {
        // Create the neutral DateTimeFormatInfo object.
        DateTimeFormatInfo dtfi =
            CultureInfo.GetCultureInfo(name).DateTimeFormat;
```

```

    // Retrieve all specific cultures of the neutral culture.
    CultureInfo[] cultures =
Array.FindAll(CultureInfo.GetCultures(CultureTypes.SpecificCultures),
                culture => culture.Name.StartsWith(name +
"--", StringComparison.OrdinalIgnoreCase));
    // Create an array of DateTimeFormatInfo properties
    PropertyInfo[] properties =
typeof(DateTimeFormatInfo).GetProperties(BindingFlags.Instance |
BindingFlags.Public);
    bool hasOneMatch = false;

    foreach (var ci in cultures)
    {
        bool match = true;
        // Get the DateTimeFormatInfo for a specific culture.
        DateTimeFormatInfo specificDtfi = ci.DateTimeFormat;
        // Compare the property values of the two.
        foreach (var prop in properties)
        {
            // We're not interested in the value of IsReadOnly.
            if (prop.Name == "IsReadOnly") continue;

            // For arrays, iterate the individual elements to see if
they are the same.
            if (prop.PropertyType.IsArray)
            {
                IList nList = (IList)prop.GetValue(dtfi, null);
                IList sList = (IList)prop.GetValue(specificDtfi, null);
                if (nList.Count != sList.Count)
                {
                    match = false;
                    Console.WriteLine("  Different n in {2} array for
{0} and {1}", name, ci.Name, prop.Name);
                    break;
                }

                for (int ctr = 0; ctr < nList.Count; ctr++)
                {
                    if (!nList[ctr].Equals(sList[ctr]))
                    {
                        match = false;
                        Console.WriteLine("  {0} value different for
{1} and {2}", prop.Name, name, ci.Name);
                        break;
                    }
                }

                if (!match) break;
            }
            // Get non-array values.
            else
            {
                Object specificValue = prop.GetValue(specificDtfi);
                Object neutralValue = prop.GetValue(dtfi);

```

```

        // Handle comparison of Calendar objects.
        if (prop.Name == "Calendar")
        {
            // The cultures have a different calendar type.
            if (specificValue.ToString() !=
neutralValue.ToString())
            {
                Console.WriteLine("  Different calendar types
for {0} and {1}", name, ci.Name);
                match = false;
                break;
            }

            if (specificValue is GregorianCalendar)
            {
                if
(((GregorianCalendar)specificValue).CalendarType !=
((GregorianCalendar)neutralValue).CalendarType)
                {
                    Console.WriteLine("  Different Gregorian
calendar types for {0} and {1}", name, ci.Name);
                    match = false;
                    break;
                }
            }
            else if (!specificValue.Equals(neutralValue))
            {
                match = false;
                Console.WriteLine("  Different {0} values for {1}
and {2}", prop.Name, name, ci.Name);
                break;
            }
        }
        if (match)
        {
            Console.WriteLine("DateTimeFormatInfo object for '{0}'
matches '{1}'",
                               name, ci.Name);
            hasOneMatch = true;
        }
    }
    if (!hasOneMatch)
        Console.WriteLine("DateTimeFormatInfo object for '{0}' --> No
Match", name);

    Console.WriteLine();
}
}

```

Instantiate a `DateTimeFormatInfo` object for the current culture

You can instantiate a `DateTimeFormatInfo` object that represents the formatting conventions of the current culture in the following ways:

- By retrieving the value of the `CurrentInfo` property. The returned `DateTimeFormatInfo` object is read-only.
- By retrieving the value of the `DateFormat` property from the `CultureInfo` object that is returned by the `CultureInfo.CurrentCulture` property. The returned `DateTimeFormatInfo` object is read-only.
- By calling the `GetInstance` method with a `CultureInfo` object that represents the current culture. The returned `DateTimeFormatInfo` object is read-only.

The following example uses each of these methods to instantiate a `DateTimeFormatInfo` object that represents the formatting conventions of the current culture. It then indicates whether the object is read-only.

C#

```
DateTimeFormatInfo dtfi;

dtfi = DateTimeFormatInfo.CurrentInfo;
Console.WriteLine(dtfi.IsReadOnly);

dtfi = CultureInfo.CurrentCulture.DateTimeFormat;
Console.WriteLine(dtfi.IsReadOnly);

dtfi = DateTimeFormatInfo.GetInstance(CultureInfo.CurrentCulture);
Console.WriteLine(dtfi.IsReadOnly);
// The example displays the following output:
//    True
//    True
//    True
```

You can create a writable `DateTimeFormatInfo` object that represents the conventions of the current culture in one of these ways:

- By retrieving a `DateTimeFormatInfo` object in any of the three previous ways and calling the `Clone` method on the returned `DateTimeFormatInfo` object. This creates a copy of the original `DateTimeFormatInfo` object, except that its `IsReadOnly` property is `false`.

- By calling the [CultureInfo.CreateSpecificCulture](#) method to create a [CultureInfo](#) object that represents the current culture, and then using its [CultureInfo.DateTimeFormat](#) property to retrieve the [DateTimeFormatInfo](#) object.

The following example illustrates each way of instantiating a read/write [DateTimeFormatInfo](#) object and displays the value of its [IsReadOnly](#) property.

C#

```
using System;
using System.Globalization;

public class InstantiateEx1
{
    public static void Main()
    {
        DateTimeFormatInfo current1 = DateTimeFormatInfo.CurrentInfo;
        current1 = (DateTimeFormatInfo)current1.Clone();
        Console.WriteLine(current1.IsReadOnly);

        CultureInfo culture2 =
CultureInfo.CreateSpecificCulture(CultureInfo.CurrentCulture.Name);
        DateTimeFormatInfo current2 = culture2.DateTimeFormat;
        Console.WriteLine(current2.IsReadOnly);
    }
}

// The example displays the following output:
//      False
//      False
```

In Windows, the user can override some of the [DateTimeFormatInfo](#) property values used in formatting and parsing operations through the [Region and Language](#) application in Control Panel. For example, a user whose culture is English (United States) might choose to display long time values using a 24-hour clock (in the format HH:mm:ss) instead of the default 12-hour clock (in the format h:mm:ss tt). The [DateTimeFormatInfo](#) objects retrieved in the ways discussed previously all reflect these user overrides. If this is undesirable, you can create a [NumberFormatInfo](#) object that does not reflect user overrides (and is also read/write instead of read-only) by calling the [CultureInfo.CultureInfo\(String, Boolean\)](#) constructor and supplying a value of `false` for the `useUserOverride` argument. The following example illustrates this for a system whose current culture is English (United States) and whose long time pattern has been changed from the default of h:mm:ss tt to HH:mm:ss.

C#

```
using System;
using System.Globalization;
```

```

public class InstantiateEx3
{
    public static void Main()
    {
        CultureInfo culture;
        DateTimeFormatInfo dtfi;

        culture = CultureInfo.CurrentCulture;
        dtfi = culture.DateTimeFormat;
        Console.WriteLine("Culture Name: {0}", culture.Name);
        Console.WriteLine("User Overrides: {0}",
culture.UseUserOverride);
        Console.WriteLine("Long Time Pattern: {0}\n",
culture.DateTimeFormat.LongTimePattern);

        culture = new CultureInfo(CultureInfo.CurrentCulture.Name, false);
        Console.WriteLine("Culture Name: {0}", culture.Name);
        Console.WriteLine("User Overrides: {0}",
culture.UseUserOverride);
        Console.WriteLine("Long Time Pattern: {0}\n",
culture.DateTimeFormat.LongTimePattern);
    }
}
// The example displays the following output:
//      Culture Name: en-US
//      User Overrides: True
//      Long Time Pattern: HH:mm:ss
//
//      Culture Name: en-US
//      User Overrides: False
//      Long Time Pattern: h:mm:ss tt

```

DateDateTimeFormatInfo and dynamic data

The culture-specific data for formatting date and time values provided by the [DateTimeFormatInfo](#) class is dynamic, just like cultural data provided by the [CultureInfo](#) class. You should not make any assumptions about the stability of values for [DateTimeFormatInfo](#) objects that are associated with particular [CultureInfo](#) objects. Only the data provided by the invariant culture and its associated [DateTimeFormatInfo](#) object is stable. Other data can change between application sessions or even while your application is running. There are four major sources of change:

- System updates. Cultural preferences such as the preferred calendar or customary date and time formats change over time. When this happens, Windows Update includes changes to the [DateTimeFormatInfo](#) property value for a particular culture.

- Replacement cultures. The [CultureAndRegionInfoBuilder](#) class can be used to replace the data of an existing culture.
- Cascading changes to property values. A number of culture-related properties can change at run time, which, in turn, causes [DateTimeFormatInfo](#) data to change. For example, the current culture can be changed either programmatically or through user action. When this happens, the [DateTimeFormatInfo](#) object returned by the [CurrentInfo](#) property changes to an object associated with the current culture. Similarly, a culture's calendar can change, which can result in changes to numerous [DateTimeFormatInfo](#) property values.
- User preferences. Users of your application might choose to override some of the values associated with the current system culture through the regional and language options in Control Panel. For example, users might choose to display the date in a different format. If the [CultureInfo.UseUserOverride](#) property is set to `true`, the properties of the [DateTimeFormatInfo](#) object is also retrieved from the user settings. If the user settings are incompatible with the culture associated with the [CultureInfo](#) object (for example, if the selected calendar is not one of the calendars indicated by the [OptionalCalendars](#) property), the results of the methods and the values of the properties are undefined.

To minimize the possibility of inconsistent data, all user-overridable properties of a [DateTimeFormatInfo](#) object are initialized when the object is created. There is still a possibility of inconsistency, because neither object creation nor the user override process is atomic and the relevant values can change during object creation. However, this situation should be extremely rare.

You can control whether user overrides are reflected in [DateTimeFormatInfo](#) objects that represent the same culture as the system culture. The following table lists the ways in which a [DateTimeFormatInfo](#) object can be retrieved and indicates whether the resulting object reflects user overrides.

[+] Expand table

| Source of CultureInfo and DateTimeFormatInfo object | Reflects user overrides |
|---|-------------------------|
| <code>CultureInfo.CurrentCulture.DateTimeFormat</code> property | Yes |
| <code>DateTimeFormatInfo.CurrentInfo</code> property | Yes |
| <code>CultureInfo.CreateSpecificCulture</code> method | Yes |
| <code>CultureInfo.GetCultureInfo</code> method | No |

| Source of CultureInfo and DateTimeFormatInfo object | Reflects user overrides |
|--|--|
| CultureInfo.CultureInfo(String) constructor | Yes |
| CultureInfo.CultureInfo(String, Boolean) constructor | Depends on value of <code>useUserOverride</code> parameter |

Unless there is a compelling reason to do otherwise, you should respect user overrides when you use the [DateTimeFormatInfo](#) object in client applications to format and parse user input or to display data. For server applications or unattended applications, you should not. However, if you are using the [DateTimeFormatInfo](#) object either explicitly or implicitly to persist date and time data in string form, you should either use a [DateTimeFormatInfo](#) object that reflects the formatting conventions of the invariant culture, or you should specify a custom date and time format string that you use regardless of culture.

Format dates and times

A [DateTimeFormatInfo](#) object is used implicitly or explicitly in all date and time formatting operations. These include calls to the following methods:

- All date and time formatting methods, such as [DateTime.ToString\(\)](#) and [DateTimeOffset.ToString\(String\)](#).
- The major composite formatting method, which is [String.Format](#).
- Other composite formatting methods, such as [Console.WriteLine\(String, Object\[\]\)](#) and [StringBuilder.AppendFormat\(String, Object\[\]\)](#).

All date and time formatting operations make use of an [IFormatProvider](#) implementation. The [IFormatProvider](#) interface includes a single method, [IFormatProvider.GetFormat\(Type\)](#). This callback method is passed a [Type](#) object that represents the type needed to provide formatting information. The method returns either an instance of that type or `null` if it cannot provide an instance of the type. .NET includes two [IFormatProvider](#) implementations for formatting dates and times:

- The [CultureInfo](#) class, which represents a specific culture (or a specific language in a specific country/region). In a date and time formatting operation, the [CultureInfo.GetFormat](#) method returns the [DateTimeFormatInfo](#) object associated with its [CultureInfo.DateTimeFormat](#) property.
- The [DateTimeFormatInfo](#) class, which provides information about the formatting conventions of its associated culture. The [DateTimeFormatInfo.GetFormat](#) method returns an instance of itself.

If an [IFormatProvider](#) implementation is not provided to a formatting method explicitly, the [CultureInfo](#) object returned by the [CultureInfo.CurrentCulture](#) property that represents the current culture is used.

The following example illustrates the relationship between the [IFormatProvider](#) interface and the [DateTimeFormatInfo](#) class in formatting operations. It defines a custom [IFormatProvider](#) implementation whose [GetFormat](#) method displays the type of the object requested by the formatting operation. If it is requesting a [DateTimeFormatInfo](#) object, the method provides the [DateTimeFormatInfo](#) object for the current culture. As the output from the example shows, the [Decimal.ToString\(IFormatProvider\)](#) method requests a [DateTimeFormatInfo](#) object to provide formatting information, whereas the [String.Format\(IFormatProvider, String, Object\[\]\)](#) method requests [NumberFormatInfo](#) and [DateTimeFormatInfo](#) objects as well as an [ICustomFormatter](#) implementation.

C#

```
using System;
using System.Globalization;

public class CurrentCultureFormatProvider : IFormatProvider
{
    public Object GetFormat(Type formatType)
    {
        Console.WriteLine("Requesting an object of type {0}",
                          formatType.Name);
        if (formatType == typeof(NumberFormatInfo))
            return NumberFormatInfo.CurrentInfo;
        else if (formatType == typeof(DateTimeFormatInfo))
            return DateTimeFormatInfo.CurrentInfo;
        else
            return null;
    }
}

public class FormatProviderEx1
{
    public static void Main()
    {
        DateTime dateValue = new DateTime(2013, 5, 28, 13, 30, 0);
        string value = dateValue.ToString("F", new
CurrentCultureFormatProvider());
        Console.WriteLine(value);
        Console.WriteLine();
        string composite = String.Format(new CurrentCultureFormatProvider(),
                                         "Date: {0:d}    Amount: {1:C}
Description: {2}",
                                         dateValue, 1264.03m, "Service
Charge");
        Console.WriteLine(composite);
        Console.WriteLine();
    }
}
```

```
        }
    }
// The example displays output like the following:
//      Requesting an object of type DateTimeFormatInfo
//      Tuesday, May 28, 2013 1:30:00 PM
//
//      Requesting an object of type ICustomFormatter
//      Requesting an object of type DateTimeFormatInfo
//      Requesting an object of type NumberFormatInfo
//      Date: 5/28/2013  Amount: $1,264.03  Description: Service Charge
```

Format strings and DateTimeFormatInfo properties

The [DateTimeFormatInfo](#) object includes three kinds of properties that are used in formatting operations with date and time values:

- Calendar-related properties. Properties such as [AbbreviatedDayNames](#), [AbbreviatedMonthNames](#), [DayNames](#), and [MonthNames](#), are associated with the calendar used by the culture, which is defined by the [Calendar](#) property. These properties are used for long date and time formats.
- Properties that produce a standards-defined result string. The [RFC1123Pattern](#), [SortableDateTimePattern](#), and [UniversalSortableDateTimePattern](#) properties contain custom format strings that produce result strings defined by international standards. These properties are read-only and cannot be modified.
- Properties that define culture-sensitive result strings. Some properties, such as [FullDateTimePattern](#) and [ShortDatePattern](#), contain [custom format strings](#) that specify the format of the result string. Others, such as [AMDesignator](#), [DateSeparator](#), [PMDesignator](#), and [TimeSeparator](#), define culture-sensitive symbols or substrings that can be included in a result string.

The [standard date and time format strings](#), such as "d", "D", "f", and "F", are aliases that correspond to particular [DateTimeFormatInfo](#) format pattern properties. Most of the [custom date and time format strings](#) are related to strings or substrings that a formatting operation inserts into the result stream. The following table lists the standard and custom date and time format specifiers and their associated [DateTimeFormatInfo](#) properties. For details about how to use these format specifiers, see [Standard Date and Time Format Strings](#) and [Custom Date and Time Format Strings](#). Note that each standard format string corresponds to a [DateTimeFormatInfo](#) property whose value is a custom date and time format string. The individual specifiers in this custom format string in turn correspond to other [DateTimeFormatInfo](#) properties. The table lists only

the [DateTimeFormatInfo](#) properties for which the standard format strings are aliases, and does not list properties that may be accessed by custom format strings assigned to those aliased properties. In addition, the table lists only custom format specifiers that correspond to [DateTimeFormatInfo](#) properties.

[+] Expand table

| Format specifier | Associated properties |
|---|--|
| "d" (short date; standard format string) | ShortDatePattern , to define the overall format of the result string. |
| "D" (long date; standard format string) | LongDatePattern , to define the overall format of the result string. |
| "f" (full date / short time; standard format string) | LongDatePattern , to define the format of the date component of the result string. ShortTimePattern , to define the format of the time component of the result string. |
| "F" (full date / long time; standard format string) | LongDatePattern , to define the format of the date component of the result string. LongTimePattern , to define the format of the time component of the result string. |
| "g" (general date / short time; standard format string) | ShortDatePattern , to define the format of the date component of the result string. ShortTimePattern , to define the format of the time component of the result string. |
| "G" (general date / long time; standard format string) | ShortDatePattern , to define the format of the date component of the result string. LongTimePattern , to define the format of the time component of the result string. |
| "M", "m" (month/day; standard format string) | MonthDayPattern , to define the overall format of the result string. |
| "O", "o" (round-trip date/time; standard format string) | None. |
| "R", "r" (RFC1123; standard format string) | RFC1123Pattern , to define a result string that conforms to the RFC 1123 standard. The property is read-only. |

| Format specifier | Associated properties |
|---|--|
| "s" (sortable date/time; standard format string) | SortableDateTimePattern , to define a result string that conforms to the ISO 8601 standard. The property is read-only. |
| "t" (short time; standard format string) | ShortTimePattern , to define the overall format of the result string. |
| "T" (long time; standard format string) | LongTimePattern , to define the overall format of the result string. |
| "u" (universal sortable date/time; standard format string) | UniversalSortableDateTimePattern , to define a result string that conforms to the ISO 8601 standard for coordinated universal time. The property is read-only. |
| "U" (universal full date/time; standard format string) | FullDateTimePattern , to define the overall format of the result string. |
| "Y", "y" (year month; standard format string) | YearMonthPattern , to define the overall format of the result string. |
| "ddd" (custom format specifier) | AbbreviatedDayNames , to include the abbreviated name of the day of the week in the result string. |
| "g", "gg" (custom format specifier) | Calls the GetEraName method to insert the era name in the result string. |
| "MMM" (custom format specifier) | AbbreviatedMonthNames , to include the abbreviated month name in the result string. |
| "MMMM" (custom format specifier) | MonthNames or MonthGenitiveNames , to include the full month name in the result string. |
| "t" (custom format specifier) | AMDesignator or PMDesignator , to include the first character of the AM/PM designator in the result string. |
| "tt" (custom format specifier) | AMDesignator or PMDesignator , to include the full AM/PM designator in the result string. |
| TimeSeparator , to include the time separator in the result string. | |
| "/" (custom format specifier) | DateSeparator , to include the date separator in the result string. |

Modify **DateTimeFormatInfo** properties

You can change the result string produced by date and time format strings by modifying the associated properties of a writable [DateTimeFormatInfo](#) object. To determine if a

[DateTimeFormatInfo](#) object is writable, use the [IsReadOnly](#) property. To customize a [DateTimeFormatInfo](#) object in this way:

1. Create a read/write copy of a [DateTimeFormatInfo](#) object whose formatting conventions you want to modify.
2. Modify the property or properties that are used to produce the desired result string. (For information about how formatting methods use [DateTimeFormatInfo](#) properties to define result strings, see the previous section, [Format strings and DateTimeFormatInfo properties](#).)
3. Use the custom [DateTimeFormatInfo](#) object you created as the [IFormatProvider](#) argument in calls to formatting methods.

There are two other ways to change the format of a result string:

- You can use the [CultureAndRegionInfoBuilder](#) class to define either a custom culture (a culture that has a unique name and that supplements existing cultures) or a replacement culture (one that is used instead of a specific culture). You can save and access this culture programmatically as you would any [CultureInfo](#) object supported by .NET.
- If the result string is not culture-sensitive and doesn't follow a predefined format, you can use a custom date and time format string. For example, if you are serializing date and time data in the format YYYYMMDDHHmmss, you can generate the result string by passing the custom format string to the [DateTime.ToString\(String\)](#) method, and you can convert the result string back to a [DateTime](#) value by calling the [DateTime.ParseExact](#) method.

Change the short date pattern

The following example changes the format of a result string produced by the "d" (short date) standard format string. It changes the associated [ShortDatePattern](#) property for the en-US or English (United States) culture from its default of "M/d/yyyy" to "yyyy"- "MM"- "dd" and uses the "d" standard format string to display the date both before and after the [ShortDatePattern](#) property is changed.

C#

```
using System;
using System.Globalization;

public class Example1
{
    public static void Main()
```

```

{
    DateTime dateValue = new DateTime(2013, 8, 18);
    CultureInfo enUS = CultureInfo.CreateSpecificCulture("en-US");
    DateTimeFormatInfo dtfi = enUS.DateTimeFormat;

    Console.WriteLine("Before modifying DateTimeFormatInfo object: ");
    Console.WriteLine("{0}: {1}\n", dtfi.ShortDatePattern,
                      dateValue.ToString("d", enUS));

    // Modify the short date pattern.
    dtfi.ShortDatePattern = "yyyy-MM-dd";
    Console.WriteLine("After modifying DateTimeFormatInfo object: ");
    Console.WriteLine("{0}: {1}", dtfi.ShortDatePattern,
                      dateValue.ToString("d", enUS));
}

// The example displays the following output:
//      Before modifying DateTimeFormatInfo object:
//      M/d/yyyy: 8/18/2013
//
//      After modifying DateTimeFormatInfo object:
//      yyyy-MM-dd: 2013-08-18

```

Change the date separator character

The following example changes the date separator character in a [DateTimeFormatInfo](#) object that represents the formatting conventions of the fr-FR culture. The example uses the "g" standard format string to display the date both before and after the [DateSeparator](#) property is changed.

C#

```

using System;
using System.Globalization;

public class Example3
{
    public static void Main()
    {
        DateTime dateValue = new DateTime(2013, 08, 28);
        CultureInfo frFR = CultureInfo.CreateSpecificCulture("fr-FR");
        DateTimeFormatInfo dtfi = frFR.DateTimeFormat;

        Console.WriteLine("Before modifying DateSeparator property: {0}",
                          dateValue.ToString("g", frFR));

        // Modify the date separator.
        dtfi.DateSeparator = "-";
        Console.WriteLine("After modifying the DateSeparator property: {0}",
                          dateValue.ToString("g", frFR));
    }
}

```

```
}
```

```
// The example displays the following output:
```

```
//      Before modifying DateSeparator property: 28/08/2013 00:00
```

```
//      After modifying the DateSeparator property: 28-08-2013 00:00
```

Change day name abbreviations and the long date pattern

In some cases, the long date pattern, which typically displays the full day and month name along with the number of the day of the month and the year, may be too long. The following example shortens the long date pattern for the en-US culture to return a one-character or two-character day name abbreviation followed by the day number, the month name abbreviation, and the year. It does this by assigning shorter day name abbreviations to the [AbbreviatedDayNames](#) array, and by modifying the custom format string assigned to the [LongDatePattern](#) property. This affects the result strings returned by the "D" and "f" standard format strings.

C#

```
using System;
using System.Globalization;

public class Example2
{
    public static void Main()
    {
        DateTime value = new DateTime(2013, 7, 9);
        CultureInfo enUS = CultureInfo.CreateSpecificCulture("en-US");
        DateTimeFormatInfo dtfi = enUS.DateTimeFormat;
        String[] formats = { "D", "F", "f" };

        // Display date before modifying properties.
        foreach (var fmt in formats)
            Console.WriteLine("{0}: {1}", fmt, value.ToString(fmt, dtfi));

        Console.WriteLine();

        // We don't want to change the FullDateTimePattern, so we need to
        // save it.
        String originalFullDateTimePattern = dtfi.FullDateTimePattern;

        // Modify day name abbreviations and long date pattern.
        dtfi.AbbreviatedDayNames = new String[] { "Su", "M", "Tu", "W",
        "Th", "F", "Sa" };
        dtfi.LongDatePattern = "ddd dd-MMM-yyyy";
        dtfi.FullDateTimePattern = originalFullDateTimePattern;
        foreach (var fmt in formats)
            Console.WriteLine("{0}: {1}", fmt, value.ToString(fmt, dtfi));
    }
}
```

```
}

// The example displays the following output:
//      D: Tuesday, July 9, 2013
//      F: Tuesday, July 9, 2013 12:00:00 AM
//      f: Tuesday, July 9, 2013 12:00 AM
//
//      D: Tu 09-Jul-2013
//      F: Tuesday, July 9, 2013 12:00:00 AM
//      f: Tu 09-Jul-2013 12:00 AM
```

Ordinarily, the change to the [LongDatePattern](#) property also affects the [FullDateTimePattern](#) property, which in turn defines the result string returned by the "F" standard format string. To preserve the original full date and time pattern, the example reassigns the original custom format string assigned to the [FullDateTimePattern](#) property after the [LongDatePattern](#) property is modified.

Change from a 12-hour clock to a 24-hour clock

For many cultures in .NET, the time is expressed by using a 12-hour clock and an AM/PM designator. The following example defines a [ReplaceWith24HourClock](#) method that replaces any time format that uses a 12-hour clock with a format that uses a 24-hour clock.

C#

```
using System;
using System.Globalization;
using System.Text.RegularExpressions;

public class Example5
{
    public static void Main()
    {
        CultureInfo enUS = CultureInfo.CreateSpecificCulture("en-US");
        DateTimeFormatInfo dtfi = enUS.DateTimeFormat;

        Console.WriteLine("Original Property Values:");
        Console.WriteLine("ShortTimePattern: " + dtfi.ShortTimePattern);
        Console.WriteLine("LongTimePattern: " + dtfi.LongTimePattern);
        Console.WriteLine("FullDateTimePattern: " +
dtfi.FullDateTimePattern);
        Console.WriteLine();

        dtfi.LongTimePattern = ReplaceWith24HourClock(dtfi.LongTimePattern);
        dtfi.ShortTimePattern =
ReplaceWith24HourClock(dtfi.ShortTimePattern);

        Console.WriteLine("Modified Property Values:");
        Console.WriteLine("ShortTimePattern: " + dtfi.ShortTimePattern);
```

```

        Console.WriteLine("LongTimePattern: " + dtfi.LongTimePattern);
        Console.WriteLine("FullDateTimePattern: " +
dtfi.FullDateTimePattern);
    }

    private static string ReplaceWith24HourClock(string fmt)
{
    string pattern = @"^(<openAMPM>\s*t+\s*)? " +
                    @"(?(openAMPM) h+(<nonHours>[^ht]+)$ " +
                    @"| \s*h+(<nonHours>[^ht]+)\s*t+)";
    return Regex.Replace(fmt, pattern, "HH${nonHours}",
                         RegexOptions.IgnorePatternWhitespace);
}
}

// The example displays the following output:
//     Original Property Values:
//     ShortTimePattern: h:mm tt
//     LongTimePattern: h:mm:ss tt
//     FullDateTimePattern: dddd, MMMM dd, yyyy h:mm:ss tt
//
//     Modified Property Values:
//     ShortTimePattern: HH:mm
//     LongTimePattern: HH:mm:ss
//     FullDateTimePattern: dddd, MMMM dd, yyyy HH:mm:ss

```

The example uses a regular expression to modify the format string. The regular expression pattern `@"^(?<openAMPM>\s*t+\s*)? (?(openAMPM) h+(<nonHours>[^ht]+)$ | \s*h+(<nonHours>[^ht]+)\s*t+)"` is defined as follows:

[\[\] Expand table](#)

| Pattern | Description |
|--|---|
| <code>^</code> | Begin the match at the beginning of the string. |
| <code>(?<openAMPM>\s*t+\s*)?</code> | Match zero or one occurrence of zero or more white-space characters, followed by the letter "t" one or more times, followed by zero or more white-space characters. This capturing group is named <code>openAMPM</code> . |
| <code>(?(openAMPM) h+(<nonHours>[^ht]+)\$</code> | If the <code>openAMPM</code> group has a match, match the letter "h" one or more times, followed by one or more characters that are neither "h" nor "t". The match ends at the end of the string. All characters captured after "h" are included in a capturing group named <code>nonHours</code> . |
| <code>&#124; \s*h+(<nonHours>[^ht]+)\s*t+</code> | If the <code>openAMPM</code> group does not have a match, match the letter "h" one or more times, followed by one or more characters that are neither "h" nor "t", followed by zero or more white-space characters. Finally, match one or more occurrences of the letter "t". All characters captured after "h" and before the white-spaces and "t" are included in a capturing group named <code>nonHours</code> . |

The `nonHours` capturing group contains the minute and possibly the second component of a custom date and time format string, along with any time separator symbols. The replacement pattern `HH${nonHours}` prepends the substring "HH" to these elements.

Display and change the era in a date

The following example adds the "g" custom format specifier to the [LongDatePattern](#) property of an object that represents the formatting conventions of the en-US culture. This addition affects the following three standard format strings:

- The "D" (long date) standard format string, which maps directly to the [LongDatePattern](#) property.
- The "f" (full date / short time) standard format string, which produces a result string that concatenates the substrings produced by the [LongDatePattern](#) and [ShortTimePattern](#) properties.
- The "F" (full date / long time) standard format string, which maps directly to the [FullDateTimePattern](#) property. Because we have not explicitly set this property value, it is generated dynamically by concatenating the [LongDatePattern](#) and [LongTimePattern](#) properties.

The example also shows how to change the era name for a culture whose calendar has a single era. In this case, the en-US culture uses the Gregorian calendar, which is represented by a [GregorianCalendar](#) object. The [GregorianCalendar](#) class supports a single era, which it names A.D. (Anno Domini). The example changes the era name to C.E. (Common Era) by replacing the "g" custom format specifier in the format string assigned to the [FullDateTimePattern](#) property with a literal string. The use of a literal string is necessary, because the era name is typically returned by the [GetEraName](#) method from private data in the culture tables supplied by either .NET or the operating system.

C#

```
using System;
using System.Globalization;

public class Example4
{
    public static void Main()
    {
        DateTime dateValue = new DateTime(2013, 5, 18, 13, 30, 0);
        String[] formats = { "D", "f", "F" };

        CultureInfo enUS = CultureInfo.CreateSpecificCulture("en-US");
```

```

        DateTimeFormatInfo dtfi = enUS.DateTimeFormat;
        String originalLongDatePattern = dtfi.LongDatePattern;

        // Display the default form of three long date formats.
        foreach (var fmt in formats)
            Console.WriteLine(dateValue.ToString(fmt, dtfi));

        Console.WriteLine();

        // Modify the long date pattern.
        dtfi.LongDatePattern = originalLongDatePattern + " g";
        foreach (var fmt in formats)
            Console.WriteLine(dateValue.ToString(fmt, dtfi));

        Console.WriteLine();

        // Change A.D. to C.E. (for Common Era)
        dtfi.LongDatePattern = originalLongDatePattern + @" 'C.E.'";
        foreach (var fmt in formats)
            Console.WriteLine(dateValue.ToString(fmt, dtfi));
    }
}

// The example displays the following output:
//      Saturday, May 18, 2013
//      Saturday, May 18, 2013 1:30 PM
//      Saturday, May 18, 2013 1:30:00 PM
//
//      Saturday, May 18, 2013 A.D.
//      Saturday, May 18, 2013 A.D. 1:30 PM
//      Saturday, May 18, 2013 A.D. 1:30:00 PM
//
//      Saturday, May 18, 2013 C.E.
//      Saturday, May 18, 2013 C.E. 1:30 PM
//      Saturday, May 18, 2013 C.E. 1:30:00 PM

```

Parse date and time strings

Parsing involves converting the string representation of a date and time to a [DateTime](#) or [DateTimeOffset](#) value. Both of these types include the `Parse`, `TryParse`, `ParseExact`, and `TryParseExact` methods to support parsing operations. The `Parse` and `TryParse` methods convert a string that can have a variety of formats, whereas `ParseExact` and `TryParseExact` require that the string have a defined format or formats. If the parsing operation fails, `Parse` and `ParseExact` throw an exception, whereas `TryParse` and `TryParseExact` return `false`.

The parsing methods implicitly or explicitly use a [DateTimeStyles](#) enumeration value to determine which style elements (such as leading, trailing, or inner white space) can be present in the string to be parsed, and how to interpret the parsed string or any missing

elements. If you don't provide a `DateTimeStyles` value when you call the `Parse` or `TryParse` method, the default is `DateTimeStyles.AllowWhiteSpaces`, which is a composite style that includes the `DateTimeStyles.AllowLeadingWhite`, `DateTimeStyles.AllowTrailingWhite`, and `DateTimeStyles.AllowInnerWhite` flags. For the `ParseExact` and `TryParseExact` methods, the default is `DateTimeStyles.None`; the input string must correspond precisely to a particular custom date and time format string.

The parsing methods also implicitly or explicitly use a `DateTimeFormatInfo` object that defines the specific symbols and patterns that can occur in the string to be parsed. If you don't provide a `DateTimeFormatInfo` object, the `DateTimeFormatInfo` object for the current culture is used by default. For more information about parsing date and time strings, see the individual parsing methods, such as `DateTime.Parse`, `DateTime.TryParse`, `DateTimeOffset.ParseExact`, and `DateTimeOffset.TryParseExact`.

The following example illustrates the culture-sensitive nature of parsing date and time strings. It tries to parse two date strings by using the conventions of the en-US, en-GB, fr-FR, and fi-FI cultures. The date that is interpreted as 8/18/2014 in the en-US culture throws a `FormatException` exception in the other three cultures because 18 is interpreted as the month number. 1/2/2015 is parsed as the second day of the first month in the en-US culture, but as the first day of the second month in the remaining cultures.

C#

```
using System;
using System.Globalization;

public class ParseEx1
{
    public static void Main()
    {
        string[] dateStrings = { "08/18/2014", "01/02/2015" };
        string[] cultureNames = { "en-US", "en-GB", "fr-FR", "fi-FI" };

        foreach (var cultureName in cultureNames)
        {
            CultureInfo culture =
CultureInfo.CreateSpecificCulture(cultureName);
            Console.WriteLine("Parsing strings using the {0} culture.",
                           culture.Name);
            foreach (var dateStr in dateStrings)
            {
                try
                {
                    Console.WriteLine(String.Format(culture,
                      "'{0}' --> {1:D}", dateStr,
                      DateTime.Parse(dateStr, culture)));
                }
            }
        }
    }
}
```

```

        catch (FormatException)
    {
        Console.WriteLine("  Unable to parse '{0}'", dateStr);
    }
}
}

// The example displays the following output:
//   Parsing strings using the en-US culture.
//     '08/18/2014' --> Monday, August 18, 2014
//     '01/02/2015' --> Friday, January 02, 2015
//   Parsing strings using the en-GB culture.
//     Unable to parse '08/18/2014'
//     '01/02/2015' --> 01 February 2015
//   Parsing strings using the fr-FR culture.
//     Unable to parse '08/18/2014'
//     '01/02/2015' --> dimanche 1 février 2015
//   Parsing strings using the fi-FI culture.
//     Unable to parse '08/18/2014'
//     '01/02/2015' --> 1. helmikuuta 2015

```

Date and time strings are typically parsed for two reasons:

- To convert user input into a date and time value.
- To round-trip a date and time value; that is, to deserialize a date and time value that was previously serialized as a string.

The following sections discuss these two operations in greater detail.

Parse user strings

When you parse date and time strings input by the user, you should always instantiate a [DateTimeFormatInfo](#) object that reflects the user's cultural settings, including any customizations the user may have made. Otherwise, the date and time object may have incorrect values. For information about how to instantiate a [DateTimeFormatInfo](#) object that reflects user cultural customizations, see the [DateTimeFormatInfo and dynamic data](#) section.

The following example illustrates the difference between a parsing operation that reflects user cultural settings and one that does not. In this case, the default system culture is en-US, but the user has used Control Panel, **Region and Language** to change the short date pattern from its default of "M/d/yyyy" to "yy/MM/dd". When the user enters a string that reflects user settings, and the string is parsed by a [DateTimeFormatInfo](#) object that also reflects user settings (overrides), the parsing operation returns a correct result. However, when the string is parsed by a [DateTimeFormatInfo](#) object that reflects standard en-US cultural settings, the parsing

method throws a `FormatException` exception because it interprets 14 as the number of the month, not the last two digits of the year.

```
C#  
  
using System;  
using System.Globalization;  
  
public class ParseEx2  
{  
    public static void Main()  
    {  
        string inputDate = "14/05/10";  
  
        CultureInfo[] cultures = { CultureInfo.GetCultureInfo("en-US"),  
                                   CultureInfo.CreateSpecificCulture("en-US")};  
  
        foreach (var culture in cultures)  
        {  
            try  
            {  
                Console.WriteLine("{0} culture reflects user overrides:  
{1}",  
                                  culture.Name, culture.UseUserOverride);  
                DateTime occasion = DateTime.Parse(inputDate, culture);  
                Console.WriteLine("'{0}' --> {1}", inputDate,  
                               occasion.ToString("D",  
CultureInfo.InvariantCulture));  
            }  
            catch (FormatException)  
            {  
                Console.WriteLine("Unable to parse '{0}'", inputDate);  
            }  
            Console.WriteLine();  
        }  
    }  
}  
// The example displays the following output:  
//      en-US culture reflects user overrides: False  
//      Unable to parse '14/05/10'  
//  
//      en-US culture reflects user overrides: True  
//      '14/05/10' --> Saturday, 10 May 2014
```

Serialize and deserialize date and time data

Serialized date and time data are expected to round-trip; that is, all serialized and deserialized values should be identical. If a date and time value represents a single moment in time, the deserialized value should represent the same moment in time

regardless of the culture or time zone of the system on which it was restored. To round-trip date and time data successfully, you must use the conventions of the invariant culture, which is returned by the [InvariantInfo](#) property, to generate and parse the data. The formatting and parsing operations should never reflect the conventions of the default culture. If you use default cultural settings, the portability of the data is strictly limited; it can be successfully serialized only on a thread whose cultural-specific settings are identical to those of the thread on which it was serialized. In some cases, this means that the data cannot even be successfully serialized and deserialized on the same system.

If the time component of a date and time value is significant, it should also be converted to UTC and serialized by using the "o" or "r" [standard format string](#). The time data can then be restored by calling a parsing method and passing it the appropriate format string along with the invariant culture as the `provider` argument.

The following example illustrates the process of round-tripping a date and time value. It serializes a date and time on a system that observes U.S. Pacific time and whose current culture is en-US.

C#

```
using System;
using System.Globalization;
using System.IO;

public class SerializeEx1
{
    public static void Main()
    {
        StreamWriter sw = new StreamWriter(@".\DateData.dat");
        // Define a date and time to serialize.
        DateTime originalDate = new DateTime(2014, 08, 18, 08, 16, 35);
        // Display information on the date and time.
        Console.WriteLine("Date to serialize: {0:F}", originalDate);
        Console.WriteLine("Current Culture: {0}",
                          CultureInfo.CurrentCulture.Name);
        Console.WriteLine("Time Zone: {0}",
                          TimeZoneInfo.Local.DisplayName);
        // Convert the date value to UTC.
        DateTime utcDate = originalDate.ToUniversalTime();
        // Serialize the UTC value.
        sw.Write(utcDate.ToString("o", DateTimeFormatInfo.InvariantInfo));
        sw.Close();
    }
}
// The example displays the following output:
//      Date to serialize: Monday, August 18, 2014 8:16:35 AM
```

```
//      Current Culture: en-US
//      Time Zone:          (UTC-08:00) Pacific Time (US & Canada)
```

It deserializes the data on a system in the Brussels, Copenhagen, Madrid and Paris time zone and whose current culture is fr-FR. The restored date is nine hours later than the original date, which reflects the time zone adjustment from eight hours behind UTC to one hour ahead of UTC. Both the original date and the restored date represent the same moment in time.

C#

```
using System;
using System.Globalization;
using System.IO;

public class SerializeEx2
{
    public static void Main()
    {
        // Open the file and retrieve the date string.
        StreamReader sr = new StreamReader(@".\DateData.dat");
        String dateValue = sr.ReadToEnd();

        // Parse the date.
        DateTime parsedDate = DateTime.ParseExact(dateValue, "o",
                                                    DateTimeFormatInfo.InvariantInfo);
        // Convert it to local time.
        DateTime restoredDate = parsedDate.ToLocalTime();
        // Display information on the date and time.
        Console.WriteLine("Deserialized date: {0:F}", restoredDate);
        Console.WriteLine("Current Culture:  {0}",
                          CultureInfo.CurrentCulture.Name);
        Console.WriteLine("Time Zone:       {0}",
                          TimeZoneInfo.Local.DisplayName);
    }
}

// The example displays the following output:
//      Deserialized date: lundi 18 août 2014 17:16:35
//      Current Culture:   fr-FR
//      Time Zone:         (UTC+01:00) Brussels, Copenhagen, Madrid, Paris
```

 Collaborate with us on
GitHub

The source for this content can
be found on GitHub, where you
can also create and review
issues and pull requests. For

.NET

 .NET feedback

.NET is an open source project.
Select a link to provide feedback:

 [Open a documentation issue](#)

more information, see [our contributor guide](#).

 [Provide product feedback](#)

NumberFormatInfo class

Article • 12/30/2023

This article provides supplementary remarks to the reference documentation for this API.

The [NumberFormatInfo](#) class contains culture-specific information that is used when you format and parse numeric values. This information includes the currency symbol, the decimal symbol, the group separator symbol, and the symbols for positive and negative signs.

Instantiate a NumberFormatInfo object

You can instantiate a [NumberFormatInfo](#) object that represents the formatting conventions of the current culture, the invariant culture, a specific culture, or a neutral culture.

Instantiate a NumberFormatInfo object for the current culture

You can instantiate a [NumberFormatInfo](#) object for the current culture in any of the following ways. In each case, the returned [NumberFormatInfo](#) object is read-only.

- By retrieving a [CultureInfo](#) object that represents the current culture from the [CultureInfo.CurrentCulture](#) property, and retrieving the [CultureInfo](#) object from its [CultureInfo.NumberFormat](#) property.
- By retrieving the [NumberFormatInfo](#) object returned by the `static` (`Shared` in Visual Basic) [CurrentInfo](#) property.
- By calling the [GetInstance](#) method with a [CultureInfo](#) object that represents the current culture.

The following example uses these three ways to create [NumberFormatInfo](#) objects that represent the formatting conventions of the current culture. It also retrieves the value of the [IsReadOnly](#) property to illustrate that each object is read-only.

C#

```
using System;
using System.Globalization;
```

```

public class InstantiateEx1
{
    public static void Main()
    {
        NumberFormatInfo current1 = CultureInfo.CurrentCulture.NumberFormat;
        Console.WriteLine(current1.IsReadOnly);

        NumberFormatInfo current2 = NumberFormatInfo.CurrentInfo;
        Console.WriteLine(current2.IsReadOnly);

        NumberFormatInfo current3 =
NumberFormatInfo.GetInstance(CultureInfo.CurrentCulture);
        Console.WriteLine(current3.IsReadOnly);
    }
}

// The example displays the following output:
//      True
//      True
//      True

```

You can create a writable [NumberFormatInfo](#) object that represents the conventions of the current culture in any of the following ways:

- By retrieving a [NumberFormatInfo](#) object in any of the ways illustrated in the previous code example, and calling the [Clone](#) method on the returned [NumberFormatInfo](#) object. This creates a copy of the original [NumberFormatInfo](#) object, except that its [IsReadOnly](#) property is `false`.
- By calling the [CultureInfo.CreateSpecificCulture](#) method to create a [CultureInfo](#) object that represents the current culture, and then using its [CultureInfo.NumberFormat](#) property to retrieve the [NumberFormatInfo](#) object.

The following example illustrates these two ways of instantiating a [NumberFormatInfo](#) object, and displays the value of its [IsReadOnly](#) property to illustrate that the object is not read-only.

C#

```

using System;
using System.Globalization;

public class InstantiateEx2
{
    public static void Main()
    {
        NumberFormatInfo current1 = NumberFormatInfo.CurrentInfo;
        current1 = (NumberFormatInfo)current1.Clone();
        Console.WriteLine(current1.IsReadOnly);

        CultureInfo culture2 =

```

```

CultureInfo.CreateSpecificCulture(CultureInfo.CurrentCulture.Name);
    NumberFormatInfo current2 = culture2.NumberFormat;
    Console.WriteLine(current2.IsReadOnly);
}
}
// The example displays the following output:
//      False
//      False

```

Note that the Windows operating system allows the user to override some of the [NumberFormatInfo](#) property values used in numeric formatting and parsing operations through the **Region and Language** item in Control Panel. For example, a user whose culture is English (United States) might choose to display currency values as 1.1 USD instead of the default of \$1.1. The [NumberFormatInfo](#) objects retrieved in the ways discussed previously all reflect these user overrides. If this is undesirable, you can create a [NumberFormatInfo](#) object that does not reflect user overrides (and that is also read/write rather than read-only) by calling the [CultureInfo.CultureInfo\(String, Boolean\)](#) constructor and supplying a value of `false` for the `useUserOverride` argument. The following example provides an illustration for a system whose current culture is English (United States) and whose currency symbol has been changed from the default of \$ to USD.

C#

```

using System;
using System.Globalization;

public class InstantiateEx3
{
    public static void Main()
    {
        CultureInfo culture;
        NumberFormatInfo nfi;

        culture = CultureInfo.CurrentCulture;
        nfi = culture.NumberFormat;
        Console.WriteLine("Culture Name: {0}", culture.Name);
        Console.WriteLine("User Overrides: {0}", culture.UseUserOverride);
        Console.WriteLine("Currency Symbol: {0}\n",
culture.NumberFormat.CurrencySymbol);

        culture = new CultureInfo(CultureInfo.CurrentCulture.Name, false);
        Console.WriteLine("Culture Name: {0}", culture.Name);
        Console.WriteLine("User Overrides: {0}", culture.UseUserOverride);
        Console.WriteLine("Currency Symbol: {0}",
culture.NumberFormat.CurrencySymbol);
    }
}
// The example displays the following output:

```

```
//      Culture Name:    en-US
//      User Overrides:  True
//      Currency Symbol: USD
//
//      Culture Name:    en-US
//      User Overrides:  False
//      Currency Symbol: $
```

If the [CultureInfo.UseUserOverride](#) property is set to `true`, the properties [CultureInfo.DateTimeFormat](#), [CultureInfo.NumberFormat](#), and [CultureInfo.TextInfo](#) are also retrieved from the user settings. If the user settings are incompatible with the culture associated with the [CultureInfo](#) object (for example, if the selected calendar is not one of the calendars listed by the [OptionalCalendars](#) property), the results of the methods and the values of the properties are undefined.

Instantiate a [NumberFormatInfo](#) object for the invariant culture

The invariant culture represents a culture that is culture-insensitive. It is based on the English language but not on any specific English-speaking country/region. Although the data of specific cultures can be dynamic and can change to reflect new cultural conventions or user preferences, the data of the invariant culture does not change. A [NumberFormatInfo](#) object that represents the formatting conventions of the invariant culture can be used for formatting operations in which result strings should not vary by culture.

You can instantiate a [NumberFormatInfo](#) object that represents the formatting conventions of the invariant culture in the following ways:

- By retrieving the value of the [InvariantInfo](#) property. The returned [NumberFormatInfo](#) object is read-only.
- By retrieving the value of the [CultureInfo.NumberFormat](#) property from the [CultureInfo](#) object that is returned by the [CultureInfo.InvariantCulture](#) property. The returned [NumberFormatInfo](#) object is read-only.
- By calling the parameterless [NumberFormatInfo](#) class constructor. The returned [NumberFormatInfo](#) object is read/write.

The following example uses each of these methods to instantiate a [NumberFormatInfo](#) object that represents the invariant culture. It then indicates whether the object is read-only,

```

using System;
using System.Globalization;

public class InstantiateEx4
{
    public static void Main()
    {
        NumberFormatInfo nfi;

        nfi = System.Globalization.NumberFormatInfo.InvariantInfo;
        Console.WriteLine(nfi.IsReadOnly);

        nfi = CultureInfo.InvariantCulture.NumberFormat;
        Console.WriteLine(nfi.IsReadOnly);

        nfi = new NumberFormatInfo();
        Console.WriteLine(nfi.IsReadOnly);
    }
}

// The example displays the following output:
//      True
//      True
//      False

```

Instantiate a `NumberFormatInfo` object for a specific culture

A specific culture represents a language that is spoken in a particular country/region. For example, en-US is a specific culture that represents the English language spoken in the United States, and en-CA is a specific culture that represents the English language spoken in Canada. You can instantiate a `NumberFormatInfo` object that represents the formatting conventions of a specific culture in the following ways:

- By calling the `CultureInfo.GetCultureInfo(String)` method and retrieving the value of the returned `CultureInfo` object's `NumberFormat` property. The returned `NumberFormatInfo` object is read-only.
- By passing a `CultureInfo` object that represents the culture whose `NumberFormatInfo` object you want to retrieve to the static `GetInstance` method. The returned `NumberFormatInfo` object is read/write.
- By calling the `CultureInfo.CreateSpecificCulture` method and retrieving the value of the returned `CultureInfo` object's `NumberFormat` property. The returned `NumberFormatInfo` object is read/write.

- By calling one of the `CultureInfo.CultureInfo` class constructors and retrieving the value of the returned `CultureInfo` object's `NumberFormat` property. The returned `NumberFormatInfo` object is read/write.

The following example uses these four ways to create a `NumberFormatInfo` object that reflects the formatting conventions of the Indonesian (Indonesia) culture. It also indicates whether each object is read-only.

C#

```
using System;
using System.Globalization;

public class InstantiateEx5
{
    public static void Main()
    {
        CultureInfo culture;
        NumberFormatInfo nfi;

        nfi = CultureInfo.GetCultureInfo("id-ID").NumberFormat;
        Console.WriteLine("Read-only: {0}", nfi.IsReadOnly);

        culture = new CultureInfo("id-ID");
        nfi = NumberFormatInfo.GetInstance(culture);
        Console.WriteLine("Read-only: {0}", nfi.IsReadOnly);

        culture = CultureInfo.CreateSpecificCulture("id-ID");
        nfi = culture.NumberFormat;
        Console.WriteLine("Read-only: {0}", nfi.IsReadOnly);

        culture = new CultureInfo("id-ID");
        nfi = culture.NumberFormat;
        Console.WriteLine("Read-only: {0}", nfi.IsReadOnly);
    }
}

// The example displays the following output:
//      Read-only: True
//      Read-only: False
//      Read-only: False
//      Read-only: False
```

Instantiate a `NumberFormatInfo` object for a neutral culture

A neutral culture represents a culture or language that is independent of a country/region. It is typically the parent of one or more specific cultures. For example, fr is a neutral culture for the French language and the parent of the fr-FR culture. You

create a [NumberFormatInfo](#) object that represents the formatting conventions of a neutral culture in the same way that you create a [NumberFormatInfo](#) object that represents the formatting conventions of a specific culture.

However, because it is independent of a specific country/region, a neutral culture lacks culture-specific formatting information. Rather than populating the [NumberFormatInfo](#) object with generic values, .NET returns a [NumberFormatInfo](#) object that reflects the formatting conventions of a specific culture that is a child of the neutral culture. For example, the [NumberFormatInfo](#) object for the neutral en culture reflects the formatting conventions of the en-US culture, and the [NumberFormatInfo](#) object for the fr culture reflects the formatting conventions of the fr-FR culture.

You can use code like the following to determine which specific culture's formatting conventions each neutral culture represents.

C#

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Globalization;
using System.Reflection;

public class InstantiateEx6
{
    public static void Main()
    {
        // Get all the neutral cultures
        List<String> names = new List<String>();
        Array.ForEach(CultureInfo.GetCultures(CultureTypes.NeutralCultures),
                     culture => names.Add(culture.Name));
        names.Sort();
        foreach (var name in names)
        {
            // Ignore the invariant culture.
            if (name == "") continue;

            ListSimilarChildCultures(name);
        }
    }

    private static void ListSimilarChildCultures(string name)
    {
        // Create the neutral NumberFormatInfo object.
        NumberFormatInfo nfi =
        CultureInfo.GetCultureInfo(name).NumberFormat;
        // Retrieve all specific cultures of the neutral culture.
        CultureInfo[] cultures =
        Array.FindAll(CultureInfo.GetCultures(CultureTypes.SpecificCultures),
                     culture => culture.Name.StartsWith(name +
```

```

"-", StringComparison.OrdinalIgnoreCase));
// Create an array of NumberFormatInfo properties
PropertyInfo[] properties =
typeof(NumberFormatInfo).GetProperties(BindingFlags.Instance |
BindingFlags.Public);
bool hasOneMatch = false;

foreach (var ci in cultures)
{
    bool match = true;
    // Get the NumberFormatInfo for a specific culture.
    NumberFormatInfo specificNfi = ci.NumberFormat;
    // Compare the property values of the two.
    foreach (var prop in properties)
    {
        // We're not interested in the value of IsReadOnly.
        if (prop.Name == "IsReadOnly") continue;

        // For arrays, iterate the individual elements to see if
they are the same.
        if (prop.PropertyType.IsArray)
        {
            IList nList = (IList)prop.GetValue(nfi, null);
            IList sList = (IList)prop.GetValue(specificNfi, null);
            if (nList.Count != sList.Count)
            {
                match = false;
                break;
            }

            for (int ctr = 0; ctr < nList.Count; ctr++)
            {
                if (!nList[ctr].Equals(sList[ctr]))
                {
                    match = false;
                    break;
                }
            }
        }
        else if
(!prop.GetValue(specificNfi).Equals(prop.GetValue(nfi)))
        {
            match = false;
            break;
        }
    }
    if (match)
    {
        Console.WriteLine("NumberFormatInfo object for '{0}' matches
'{1}'",
                           name, ci.Name);
        hasOneMatch = true;
    }
}
if (!hasOneMatch)

```

```
        Console.WriteLine("NumberFormatInfo object for '{0}' --> No  
Match", name);  
  
        Console.WriteLine();  
    }  
}
```

Dynamic data

The culture-specific data for formatting numeric values provided by the [NumberFormatInfo](#) class is dynamic, just like the cultural data provided by the [CultureInfo](#) class. You should not make any assumptions about the stability of values for [NumberFormatInfo](#) objects that are associated with particular [CultureInfo](#) objects. Only the data provided by the invariant culture and its associated [NumberFormatInfo](#) object is stable. Other data can change between application sessions, or even within a single session, for the following reasons:

- **System updates.** Cultural preferences such as the currency symbol or currency formats change over time. When this happens, Windows Update includes changes to the [NumberFormatInfo](#) property value for a particular culture.
- **Replacement cultures.** The [CultureAndRegionInfoBuilder](#) class can be used to replace the data of an existing culture.
- **Cascading changes to property values.** A number of culture-related properties can change at run time, which, in turn, causes [NumberFormatInfo](#) data to change. For example, the current culture can be changed either programmatically or through user action. When this happens, the [NumberFormatInfo](#) object returned by the [CurrentInfo](#) property changes to an object associated with the current culture.
- **User preferences.** Users of your application might override some of the values associated with the current system culture through the region and language options in Control Panel. For example, users might choose a different currency symbol or a different decimal separator symbol. If the [CultureInfo.UseUserOverride](#) property is set to `true` (its default value), the properties of the [NumberFormatInfo](#) object are also retrieved from the user settings.

All user-overridable properties of a [NumberFormatInfo](#) object are initialized when the object is created. There is still a possibility of inconsistency, because neither object creation nor the user override process is atomic, and the relevant values may change during object creation. However, these inconsistencies should be extremely rare.

You can control whether user overrides are reflected in [NumberFormatInfo](#) objects that represent the same culture as the current culture. The following table lists the ways in which a [NumberFormatInfo](#) object can be retrieved and indicates whether the resulting object reflects user overrides.

 Expand table

| Source of CultureInfo and NumberFormatInfo object | Reflects user overrides |
|---|--|
| <code>CultureInfo.CurrentCulture.NumberFormat</code> property | Yes |
| <code>NumberFormatInfo.CurrentInfo</code> property | Yes |
| <code>CultureInfo.CreateSpecificCulture</code> method | Yes |
| <code>CultureInfo.GetCultureInfo</code> method | No |
| <code>CultureInfo(String)</code> constructor | Yes |
| <code>CultureInfo.CultureInfo(String, Boolean)</code> constructor | Depends on value of <code>useUserOverride</code> parameter |

Unless there is a compelling reason to do otherwise, you should respect user overrides when you use the [NumberFormatInfo](#) object in client applications to format and parse user input or to display numeric data. For server applications or unattended applications, you should not respect user overrides. However, if you are using the [NumberFormatInfo](#) object either explicitly or implicitly to persist numeric data in string form, you should either use a [NumberFormatInfo](#) object that reflects the formatting conventions of the invariant culture, or you should specify a custom numeric format string that you use regardless of culture.

IFormatProvider, NumberFormatInfo, and numeric formatting

A [NumberFormatInfo](#) object is used implicitly or explicitly in all numeric formatting operations. These include calls to the following methods:

- All numeric formatting methods, such as [Int32.ToString](#), [Double.ToString](#), and [Convert.ToString\(Int32\)](#).
- The major composite formatting method, [String.Format](#).
- Other composite formatting methods, such as [Console.WriteLine\(String, Object\[\]\)](#) and [StringBuilder.AppendFormat\(String, Object\[\]\)](#).

All numeric formatting operations make use of an [IFormatProvider](#) implementation. The [IFormatProvider](#) interface includes a single method, [GetFormat\(Type\)](#). This is a callback method that is passed a [Type](#) object that represents the type needed to provide formatting information. The method is responsible for returning either an instance of that type or `null`, if it cannot provide an instance of the type. .NET provides two [IFormatProvider](#) implementations for formatting numbers:

- The [CultureInfo](#) class, which represents a specific culture (or a specific language in a specific country/region). In a numeric formatting operation, the [CultureInfo.GetFormat](#) method returns the [NumberFormatInfo](#) object associated with its [CultureInfo.NumberFormat](#) property.
- The [NumberFormatInfo](#) class, which provides information about the formatting conventions of its associated culture. The [NumberFormatInfo.GetFormat](#) method returns an instance of itself.

If an [IFormatProvider](#) implementation is not provided to a formatting method explicitly, a [CultureInfo](#) object returned by the [CultureInfo.CurrentCulture](#) property that represents the current culture is used.

The following example illustrates the relationship between the [IFormatProvider](#) interface and the [NumberFormatInfo](#) class in formatting operations by defining a custom [IFormatProvider](#) implementation. Its [GetFormat](#) method displays the type name of the object requested by the formatting operation. If the interface is requesting a [NumberFormatInfo](#) object, this method provides the [NumberFormatInfo](#) object for the current culture. As the output from the example shows, the [Decimal.ToString\(IFormatProvider\)](#) method requests a [NumberFormatInfo](#) object to provide formatting information, whereas the [String.Format\(IFormatProvider, String, Object\[\]\)](#) method requests [NumberFormatInfo](#) and [DateTimeFormatInfo](#) objects as well as an [ICustomFormatter](#) implementation.

C#

```
using System;
using System.Globalization;

public class CurrentCultureFormatProvider : IFormatProvider
{
    public Object GetFormat(Type formatType)
    {
        Console.WriteLine("Requesting an object of type {0}",
            formatType.Name);
        if (formatType == typeof(NumberFormatInfo))
            return NumberFormatInfo.CurrentInfo;
        else if (formatType == typeof(DateTimeFormatInfo))
            return DateTimeFormatInfo.CurrentInfo;
```

```

        else
            return null;
    }

public class FormatProviderEx
{
    public static void Main()
    {
        Decimal amount = 1203.541m;
        string value = amount.ToString("C2", new
CurrentCultureFormatProvider());
        Console.WriteLine(value);
        Console.WriteLine();
        string composite = String.Format(new CurrentCultureFormatProvider(),
                                         "Date: {0}    Amount: {1}
Description: {2}",
                                         DateTime.Now, 1264.03m, "Service
Charge");
        Console.WriteLine(composite);
        Console.WriteLine();
    }
}

// The example displays output like the following:
// Requesting an object of type NumberFormatInfo
// $1,203.54
//
// Requesting an object of type ICustomFormatter
// Requesting an object of type DateTimeFormatInfo
// Requesting an object of type NumberFormatInfo
// Date: 11/15/2012 2:00:01 PM    Amount: 1264.03    Description: Service
Charge

```

If an [IFormatProvider](#) implementation is not explicitly provided in a numeric formatting method call, the method calls the `CultureInfo.CurrentCulture.GetFormat` method, which returns the [NumberFormatInfo](#) object that corresponds to the current culture.

Format strings and NumberFormatInfo properties

Every formatting operation uses either a standard or a custom numeric format string to produce a result string from a number. In some cases, the use of a format string to produce a result string is explicit, as in the following example. This code calls the `Decimal.ToString(IFormatProvider)` method to convert a [Decimal](#) value to a number of different string representations by using the formatting conventions of the en-US culture.

```

using System;
using System.Globalization;

public class PropertiesEx1
{
    public static void Main()
    {
        string[] formatStrings = { "C2", "E1", "F", "G3", "N",
                                   "#,##0.000", "0,000,000,000.0##" };
        CultureInfo culture = CultureInfo.CreateSpecificCulture("en-US");
        Decimal[] values = { 1345.6538m, 1921651.16m };

        foreach (var value in values)
        {
            foreach (var formatString in formatStrings)
            {
                string resultString = value.ToString(formatString, culture);
                Console.WriteLine("{0,-18} --> {1}", formatString,
resultString);
            }
            Console.WriteLine();
        }
    }
}

// The example displays the following output:
//      C2          --> $1,345.65
//      E1          --> 1.3E+003
//      F           --> 1345.65
//      G3          --> 1.35E+03
//      N           --> 1,345.65
//      #,##0.000    --> 1,345.654
//      0,000,000,000.0## --> 0,000,001,345.654
//
//      C2          --> $1,921,651.16
//      E1          --> 1.9E+006
//      F           --> 1921651.16
//      G3          --> 1.92E+06
//      N           --> 1,921,651.16
//      #,##0.000    --> 1,921,651.160
//      0,000,000,000.0## --> 0,001,921,651.16

```

In other cases, the use of a format string is implicit. For example, in the following method calls to the default or parameterless `Decimal.ToString()` method, the value of the `Decimal` instance is formatted by using the general ("G") format specifier and the conventions of the current culture, which in this case is the en-US culture.

C#

```

using System;

public class PropertiesEx2

```

```

{
    public static void Main()
    {
        Decimal[] values = { 1345.6538m, 1921651.16m };

        foreach (var value in values)
        {
            string resultString = value.ToString();
            Console.WriteLine(resultString);
            Console.WriteLine();
        }
    }
}

// The example displays the following output:
//      1345.6538
//
//      1921651.16

```

Each standard numeric format string uses one or more [NumberFormatInfo](#) properties to determine the pattern or the symbols used in the result string. Similarly, each custom numeric format specifier except "0" and "#" insert symbols in the result string that are defined by [NumberFormatInfo](#) properties. The following table lists the standard and custom numeric format specifiers and their associated [NumberFormatInfo](#) properties. To change the appearance of the result string for a particular culture, see the [Modify NumberFormatInfo properties](#) section. For details about the use of these format specifiers, see [Standard Numeric Format Strings](#) and [Custom Numeric Format Strings](#).

[] [Expand table](#)

| Format specifier | Associated properties |
|--|--|
| "C" or "c" (currency format specifier) | CurrencyDecimalDigits , to define the default number of fractional digits. CurrencyDecimalSeparator , to define the decimal separator symbol. CurrencyGroupSeparator , to define the group or thousands separator. CurrencyGroupSizes , to define the sizes of integral groups. |
| | CurrencyNegativePattern , to define the pattern of negative currency values. |
| | CurrencyPositivePattern , to define the pattern of positive currency values. |

| Format specifier | Associated properties |
|---|---|
| | <p>CurrencySymbol, to define the currency symbol.</p> <p>NegativeSign, to define the negative sign symbol.</p> |
| "D" or "d" (decimal format specifier) | <p>NegativeSign, to define the negative sign symbol.</p> |
| "E" or "e" (exponential or scientific format specifier) | <p>NegativeSign, to define the negative sign symbol in the mantissa and exponent.</p> <p>NumberDecimalSeparator, to define the decimal separator symbol.</p> <p>PositiveSign, to define the positive sign symbol in the exponent.</p> |
| "F" or "f" (fixed-point format specifier) | <p>NegativeSign, to define the negative sign symbol.</p> <p>NumberDecimalDigits, to define the default number of fractional digits.</p> <p>NumberDecimalSeparator, to define the decimal separator symbol.</p> |
| "G" or "g" (general format specifier) | <p>NegativeSign, to define the negative sign symbol.</p> <p>NumberDecimalSeparator, to define the decimal separator symbol.</p> <p>PositiveSign, to define the positive sign symbol for result strings in exponential format.</p> |
| "N" or "n" (number format specifier) | <p>NegativeSign, to define the negative sign symbol.</p> <p>NumberDecimalDigits, to define the default number of fractional digits.</p> <p>NumberDecimalSeparator, to define the decimal separator symbol.</p> <p>NumberGroupSeparator, to define the group separator (thousands) symbol.</p> <p>NumberGroupSizes, to define the number of integral digits in a group.</p> <p>NumberNegativePattern, to define the format of negative values.</p> |
| "P" or "p" (percent format specifier) | <p>NegativeSign, to define the negative sign symbol.</p> |

| | |
|--|---|
| Format specifier | <p>PercentDecimalDigits, to define the default number of fractional digits.</p> <p>PercentDecimalSeparator, to define the decimal separator symbol.</p> <p>PercentGroupSeparator, to define the group separator symbol.</p> <p>PercentGroupSizes, to define the number of integral digits in a group.</p> <p>PercentNegativePattern, to define the placement of the percent symbol and the negative symbol for negative values.</p> <p>PercentPositivePattern, to define the placement of the percent symbol for positive values.</p> <p>PercentSymbol, to define the percent symbol.</p> |
| "R" or "r" (round-trip format specifier) | <p>NegativeSign, to define the negative sign symbol.</p> <p>NumberDecimalSeparator, to define the decimal separator symbol.</p> <p>PositiveSign, to define the positive sign symbol in an exponent.</p> |
| "X" or "x" (hexadecimal format specifier) | None. |
| "." (decimal point custom format specifier) | NumberDecimalSeparator , to define the decimal separator symbol. |
| "," (group separator custom format specifier) | NumberGroupSeparator , to define the group (thousands) separator symbol. |
| "%" (percentage placeholder custom format specifier) | PercentSymbol , to define the percent symbol. |
| "‰" (per mille placeholder custom format specifier) | PerMilleSymbol , to define the per mille symbol. |
| "E" (exponential notation custom format specifier) | <p>NegativeSign, to define the negative sign symbol in the mantissa and exponent.</p> <p>PositiveSign, to define the positive sign symbol in the exponent.</p> |

Note that the [NumberFormatInfo](#) class includes a [NativeDigits](#) property that specifies the base 10 digits used by a specific culture. However, the property is not used in formatting operations; only the Basic Latin digits 0 (U+0030) through 9 (U+0039) are

used in the result string. In addition, for [Single](#) and [Double](#) values of `NaN`, `PositiveInfinity`, and `NegativeInfinity`, the result string consists exclusively of the symbols defined by the [NaNSymbol](#), [PositiveInfinitySymbol](#), and [NegativeInfinitySymbol](#) properties, respectively.

Modify NumberFormatInfo properties

You can modify the properties of a [NumberFormatInfo](#) object to customize the result string produced in a numeric formatting operation. To do this:

1. Create a read/write copy of a [NumberFormatInfo](#) object whose formatting conventions you want to modify. For more information, see the [Instantiate a NumberFormatInfo object](#) section.
2. Modify the property or properties that are used to produce the desired result string. For information about how formatting methods use [NumberFormatInfo](#) properties to define result strings, see the [Format strings and NumberFormatInfo properties](#) section.
3. Use the custom [NumberFormatInfo](#) object as the [IFormatProvider](#) argument in calls to formatting methods.

ⓘ Note

Instead of dynamically modifying a culture's property values each time an application is started, you can use the [CultureAndRegionInfoBuilder](#) class to define either a custom culture (a culture that has a unique name and that supplements existing cultures) or a replacement culture (one that is used instead of a specific culture).

The following sections provide some examples.

Modify the currency symbol and pattern

The following example modifies a [NumberFormatInfo](#) object that represents the formatting conventions of the en-US culture. It assigns the ISO-4217 currency symbol to the [CurrencySymbol](#) property and defines a pattern for currency values that consists of the currency symbol followed by a space and a numeric value.

C#

```

using System;
using System.Globalization;

public class Example
{
    public static void Main()
    {
        // Retrieve a writable NumberFormatInfo object.
        CultureInfo enUS = CultureInfo.CreateSpecificCulture("en-US");
        NumberFormatInfo nfi = enUS.NumberFormat;

        // Use the ISO currency symbol instead of the native currency
        symbol.
        nfi.CurrencySymbol = (new RegionInfo(enUS.Name)).ISOCurrencySymbol;
        // Change the positive currency pattern to <code><space><value>.
        nfi.CurrencyPositivePattern = 2;
        // Change the negative currency pattern to <code><space><sign>
        <value>.
        nfi.CurrencyNegativePattern = 12;

        // Produce the result strings by calling ToString.
        Decimal[] values = { 1065.23m, 19.89m, -.03m, -175902.32m };
        foreach (var value in values)
            Console.WriteLine(value.ToString("C", enUS));

        Console.WriteLine();

        // Produce the result strings by calling a composite formatting
        method.
        foreach (var value in values)
            Console.WriteLine(String.Format(enUS, "{0:C}", value));
    }
}

// The example displays the following output:
//      USD 1,065.23
//      USD 19.89
//      USD -0.03
//      USD -175,902.32
//
//      USD 1,065.23
//      USD 19.89
//      USD -0.03
//      USD -175,902.32

```

Format a national identification number

Many national identification numbers consist exclusively of digits and so can easily be formatted by modifying the properties of a [NumberFormatInfo](#) object. For example, a social security number in the United States consists of 9 digits arranged as follows: xxx-

xx-xxxx. The following example assumes that social security numbers are stored as integer values and formats them appropriately.

C#

```
using System;
using System.Globalization;

public class CustomizeSSNEx
{
    public static void Main()
    {
        // Instantiate a read-only NumberFormatInfo object.
        CultureInfo enUS = CultureInfo.CreateSpecificCulture("en-US");
        NumberFormatInfo nfi = enUS.NumberFormat;

        // Modify the relevant properties.
        nfi.NumberGroupSeparator = "-";
        nfi.NumberGroupSizes = new int[] { 3, 2, 4 };
        nfi.NumberDecimalDigits = 0;

        int[] ids = { 111223333, 999776666 };

        // Produce the result string by calling ToString.
        foreach (var id in ids)
            Console.WriteLine(id.ToString("N", enUS));

        Console.WriteLine();

        // Produce the result string using composite formatting.
        foreach (var id in ids)
            Console.WriteLine(String.Format(enUS, "{0:N}", id));
    }
}

// The example displays the following output:
//      1112-23-333
//      9997-76-666
//
//      1112-23-333
//      9997-76-666
```

Parse numeric strings

Parsing involves converting the string representation of a number to a number. Each numeric type in .NET includes two overloaded parsing methods: `Parse` and `TryParse`. The `Parse` method converts a string to a number and throws an exception if the conversion fails. The `TryParse` method converts a string to a number, assigns the number to an `out` argument, and returns a `Boolean` value that indicates whether the conversion succeeded.

The parsing methods implicitly or explicitly use a [NumberStyles](#) enumeration value to determine what style elements (such as group separators, a decimal separator, or a currency symbol) can be present in a string if the parsing operation is to succeed. If a [NumberStyles](#) value is not provided in the method call, the default is a [NumberStyles](#) value that includes the [Float](#) and [AllowThousands](#) flags, which specifies that the parsed string can include group symbols, a decimal separator, a negative sign, and white-space characters, or it can be the string representation of a number in exponential notation.

The parsing methods also implicitly or explicitly use a [NumberFormatInfo](#) object that defines the specific symbols and patterns that can occur in the string to be parsed. If a [NumberFormatInfo](#) object is not provided, the default is the [NumberFormatInfo](#) for the current culture. For more information about parsing, see the individual parsing methods, such as [Int16.Parse\(String\)](#), [Int32.Parse\(String, NumberStyles\)](#), [Int64.Parse\(String, IFormatProvider\)](#), [Decimal.Parse\(String, NumberStyles, IFormatProvider\)](#), [Double.TryParse\(String, Double\)](#), and [BigInteger.TryParse\(String, NumberStyles, IFormatProvider, BigInteger\)](#).

The following example illustrates the culture-sensitive nature of parsing strings. It tries to parse a string that include thousands separators by using the conventions of the en-US, fr-FR, and invariant cultures. A string that includes the comma as a group separator and the period as a decimal separator fails to parse in the fr-FR culture, and a string with white space as a group separator and a comma as a decimal separator fails to parse in the en-US and invariant cultures.

C#

```
using System;
using System.Globalization;

public class ParseEx1
{
    public static void Main()
    {
        String[] values = { "1,034,562.91", "9 532 978,07" };
        String[] cultureNames = { "en-US", "fr-FR", "" };

        foreach (var value in values)
        {
            foreach (var cultureName in cultureNames)
            {
                CultureInfo culture =
CultureInfo.CreateSpecificCulture(cultureName);
                String name = culture.Name == "" ? "Invariant" :
culture.Name;
                try
                {
                    Decimal amount = Decimal.Parse(value, culture);
                }
            }
        }
    }
}
```

```

                Console.WriteLine("'{0}' --> {1} ({2})", value, amount,
name);
            }
            catch (FormatException)
            {
                Console.WriteLine("'{0}': FormatException ({1})",
value, name);
            }
        }
        Console.WriteLine();
    }
}
// The example displays the following output:
//      '1,034,562.91' --> 1034562.91 (en-US)
//      '1,034,562.91': FormatException (fr-FR)
//      '1,034,562.91' --> 1034562.91 (Invariant)
//
//      '9 532 978,07': FormatException (en-US)
//      '9 532 978,07' --> 9532978.07 (fr-FR)
//      '9 532 978,07': FormatException (Invariant)

```

Parsing generally occurs in two contexts:

- As an operation that is designed to convert user input into a numeric value.
- As an operation that is designed to round-trip a numeric value; that is, to deserialize a numeric value that was previously serialized as a string.

The following sections discuss these two operations in greater detail.

Parse user strings

When you are parsing numeric strings input by the user, you should always instantiate a [NumberFormatInfo](#) object that reflects the user's cultural settings. For information about how to instantiate a [NumberFormatInfo](#) object that reflects user customizations, see the [Dynamic data](#) section.

The following example illustrates the difference between a parsing operation that reflects user cultural settings and one that does not. In this case, the default system culture is en-US, but the user has defined "," as the decimal symbol and "." as the group separator in Control Panel, **Region and Language**. Ordinarily, these symbols are reversed in the default en-US culture. When the user enters a string that reflects user settings, and the string is parsed by a [NumberFormatInfo](#) object that also reflects user settings (overrides), the parsing operation returns a correct result. However, when the string is parsed by a [NumberFormatInfo](#) object that reflects standard en-US cultural

settings, it mistakes the comma symbol for a group separator and returns an incorrect result.

```
C#  
  
using System;  
using System.Globalization;  
  
public class ParseUserEx  
{  
    public static void Main()  
    {  
        CultureInfo stdCulture = CultureInfo.GetCultureInfo("en-US");  
        CultureInfo custCulture = CultureInfo.CreateSpecificCulture("en-US");  
  
        String value = "310,16";  
        try  
        {  
            Console.WriteLine("{0} culture reflects user overrides: {1}",  
                stdCulture.Name, stdCulture.UseUserOverride);  
            Decimal amount = Decimal.Parse(value, stdCulture);  
            Console.WriteLine("'{0}' --> {1}", value,  
                amount.ToString(CultureInfo.InvariantCulture));  
        }  
        catch (FormatException)  
        {  
            Console.WriteLine("Unable to parse '{0}'", value);  
        }  
        Console.WriteLine();  
  
        try  
        {  
            Console.WriteLine("{0} culture reflects user overrides: {1}",  
                custCulture.Name,  
                custCulture.UseUserOverride);  
            Decimal amount = Decimal.Parse(value, custCulture);  
            Console.WriteLine("'{0}' --> {1}", value,  
                amount.ToString(CultureInfo.InvariantCulture));  
        }  
        catch (FormatException)  
        {  
            Console.WriteLine("Unable to parse '{0}'", value);  
        }  
    }  
}  
// The example displays the following output:  
//      en-US culture reflects user overrides: False  
//      '310,16' --> 31016  
//  
//      en-US culture reflects user overrides: True  
//      '310,16' --> 310.16
```

Serialize and deserialize numeric data

When numeric data is serialized in string format and later deserialized and parsed, the strings should be generated and parsed by using the conventions of the invariant culture. The formatting and parsing operations should never reflect the conventions of a specific culture. If culture-specific settings are used, the portability of the data is strictly limited; it can be successfully deserialized only on a thread whose culture-specific settings are identical to those of the thread on which it was serialized. In some cases, this means that the data cannot even be successfully deserialized on the same system on which it was serialized.

The following example illustrates what can happen when this principle is violated. Floating-point values in an array are converted to strings when the current thread uses the culture-specific settings of the en-US culture. The data is then parsed by a thread that uses the culture-specific settings of the pt-BR culture. In this case, although each parsing operation succeeds, the data doesn't round-trip successfully and data corruption occurs. In other cases, a parsing operation could fail and a [FormatException](#) exception could be thrown.

C#

```
using System;
using System.Collections.Generic;
using System.Globalization;
using System.IO;
using System.Threading;

public class ParsePersistedEx
{
    public static void Main()
    {
        CultureInfo.CurrentCulture = CultureInfo.CreateSpecificCulture("en-US");
        PersistData();

        CultureInfo.CurrentCulture = CultureInfo.CreateSpecificCulture("pt-BR");
        RestoreData();
    }

    private static void PersistData()
    {
        // Define an array of floating-point values.
        Double[] values = { 160325.972, 8631.16, 1.304e5, 98017554.385,
                           8.5938287084321676e94 };
        Console.WriteLine("Original values: ");
        foreach (var value in values)
            Console.WriteLine(value.ToString("R",
                CultureInfo.InvariantCulture));
    }
}
```

```

// Serialize an array of doubles to a file
StreamWriter sw = new StreamWriter(@".\NumericData.bin");
for (int ctr = 0; ctr < values.Length; ctr++)
{
    sw.Write(values[ctr].ToString("R"));
    if (ctr < values.Length - 1) sw.Write("|");
}
sw.Close();
Console.WriteLine();
}

private static void RestoreData()
{
    // Deserialize the data
    StreamReader sr = new StreamReader(@".\NumericData.bin");
    String data = sr.ReadToEnd();
    sr.Close();

    String[] stringValues = data.Split('|');
    List<Double> newValueList = new List<Double>();

    foreach (var stringValue in stringValues)
    {
        try
        {
            newValueList.Add(Double.Parse(stringValue));
        }
        catch (FormatException)
        {
            newValueList.Add(Double.NaN);
        }
    }

    Console.WriteLine("Restored values:");
    foreach (var newValue in newValueList)
        Console.WriteLine(newValue.ToString("R",
NumberFormatInfo.InvariantInfo));
}

// The example displays the following output:
//      Original values:
//      160325.972
//      8631.16
//      130400
//      98017554.385
//      8.5938287084321671E+94
//
//      Restored values:
//      160325972
//      863116
//      130400
//      98017554385
//      8.5938287084321666E+110

```

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

.NET is an open source project.
Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

PersianCalendar class

Article • 12/30/2023

This article provides supplementary remarks to the reference documentation for this API.

The Persian calendar is used in most countries/regions where Persian is spoken, although some regions use different month names. The Persian calendar is the official calendar of Iran and Afghanistan, and it is one of the alternative calendars in regions such as Kazakhstan and Tajikistan.

ⓘ Note

For information about using the [PersianCalendar](#) class and the other calendar classes in .NET, see [Working with Calendars](#).

The Persian calendar is a solar Hijri calendar, and starts from the year of the Hijra, which corresponds to 622 C.E. the year when Muhammad (PBUH) migrated from Mecca to Medina.

The Persian calendar is based on a solar year and is approximately 365 days long. A year cycles through four seasons, and a new year begins when the sun appears to cross the equator from the southern hemisphere to the northern hemisphere as viewed from the center of the Earth. The new year marks the first day of the month of Farvardeen, which is the first day of spring in the northern hemisphere. For example, the date March 21, 2002 C.E. corresponds to the first day of the month of Farvardeen in the year 1381 Anno Persico.

Each of the first six months in the Persian calendar has 31 days, each of the next five months has 30 days, and the last month has 29 days in a common year and 30 days in a leap year. A leap year is a year that, when divided by 33, has a remainder of 1, 5, 9, 13, 17, 22, 26, or 30. For example, the year 1370 is a leap year because dividing it by 33 yields a remainder of 17. There are approximately eight leap years in every 33-year cycle.

The PersianCalendar class and .NET Framework versions

Starting with .NET Framework 4.6, the [PersianCalendar](#) class uses the Hijri solar astronomical algorithm rather than an observational algorithm to calculate dates. This makes the [PersianCalendar](#) implementation consistent with the Persian calendar in use

in Iran and Afghanistan, the two countries in which the Persian calendar is in most widespread use. The change affects all apps running on the .NET Framework 4 or later if the .NET Framework 4.6 is installed.

As a result of the changed algorithm:

- The two algorithms should return identical results when converting dates between 1800 and 2123 in the Gregorian calendar.
- The two algorithms may return different results when converting dates before 1800 and after 2123 in the Gregorian calendar.
- The [MinSupportedDateTime](#) property value has changed from March 21, 0622 in the Gregorian calendar to March 22, 0622 in the Gregorian calendar.
- The [MaxSupportedDateTime](#) property value has changed from the 10th day of the 10th month of the year 9378 in the Persian calendar to the 13th day of the 10th month of the year 9378 in the Persian calendar.
- The [IsLeapYear](#) method may return a different result than it did previously.

Use the [PersianCalendar](#) class

Applications use a [PersianCalendar](#) object to calculate dates in the Persian calendar or convert Persian dates to and from Gregorian dates.

You cannot use a [PersianCalendar](#) object as the default calendar for a culture. The default calendar is specified by the [CultureInfo.Calendar](#) property and must be one of the calendars returned by the [CultureInfo.OptionalCalendars](#) property. Currently, the [PersianCalendar](#) class is not an optional calendar for any culture supported by the [CultureInfo](#) class and consequently cannot be a default calendar.

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



.NET feedback

.NET is an open source project.
Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

RegionInfo class

Article • 12/30/2023

This article provides supplementary remarks to the reference documentation for this API.

Unlike the [CultureInfo](#) class, the [RegionInfo](#) class does not represent user preferences and does not depend on the user's language or culture.

Names associated with a RegionInfo object

The name of a [RegionInfo](#) object is one of the two-letter codes defined in ISO 3166 for country/region. Case is not significant. The [Name](#), [TwoLetterISORegionName](#), and [ThreeLetterISORegionName](#) properties return the appropriate codes in uppercase. For the current list of [RegionInfo](#) names, see [ISO 3166: Country codes ↗](#).

Instantiate a RegionInfo object

To instantiate a [RegionInfo](#) object, you pass the [RegionInfo\(String\)](#) constructor either a two-letter region name, such as "US" for the United States, or the name of a specific culture, such as "en-US" for English (United States). However, we recommend that you use a specific culture name instead of a two-letter region name, because a [RegionInfo](#) object is not completely language-independent. Several [RegionInfo](#) properties, including [DisplayName](#), [NativeName](#), and [CurrencyNativeName](#), depend on culture names.

The following example illustrates the difference in [RegionInfo](#) property values for three objects that represent Belgium. The first is instantiated from a region name (`BE`) only, while the second and third are instantiated from culture names (`fr-BE` for French (Belgium) and `nl-BE` for Dutch (Belgium), respectively). The example uses reflection to retrieve the property values of each [RegionInfo](#) object.

C#

```
using System;
using System.Globalization;
using System.Reflection;

public class Example
{
    public static void Main()
    {
```

```

        // Instantiate three Belgian RegionInfo objects.
        RegionInfo BE = new RegionInfo("BE");
        RegionInfo frBE = new RegionInfo("fr-BE");
        RegionInfo nlBE = new RegionInfo("nl-BE");

        RegionInfo[] regions = { BE, frBE, nlBE };
        PropertyInfo[] props =
typeof(RegionInfo).GetProperties(BindingFlags.Instance |
BindingFlags.Public);

        Console.WriteLine("{0,-30}{1,18}{2,18}{3,18}\n",
                        "RegionInfo Property", "BE", "fr-BE", "nl-BE");
        foreach (var prop in props)
        {
            Console.Write("{0,-30}", prop.Name);
            foreach (var region in regions)
                Console.Write("{0,18}", prop.GetValue(region, null));

            Console.WriteLine();
        }
    }
}

// The example displays the following output:
//      RegionInfo Property           BE          fr-BE
nl-BE
//
//      Name                      BE          fr-BE
nl-BE
//      EnglishName               Belgium     Belgium
Belgium
//      DisplayName               Belgium     Belgium
Belgium
//      NativeName                België     Belgique
België
//      TwoLetterISORegionName    BE          BE
BE
//      ThreeLetterISORegionName BEL         BEL
BEL
//      ThreeLetterWindowsRegionName BEL         BEL
BEL
//      IsMetric                  True        True
True
//      GeoId                     21          21
21
//      CurrencyEnglishName      Euro        Euro
Euro
//      CurrencyNativeName       euro        euro
euro
//      CurrencySymbol            €           €
€
//      ISOCurrencySymbol        EUR        EUR
EUR

```

In scenarios such as the following, use culture names instead of country/region names when you instantiate a [RegionInfo](#) object:

- When the language name is of primary importance. For example, for the `es-US` culture name, you'll probably want your application to display "Estados Unidos" instead of "United States". Using the country/region name (`us`) alone yields "United States" regardless of the language, so you should work with the culture name instead.
- When script differences must be considered. For example, the country/region `AZ` deals with Azerbaijani cultures that have the names `az-Latn-AZ` and `az-Cyr1-AZ`, and the Latin and Cyrillic scripts can be very different for this country/region.
- When maintenance of detail is important. The values returned by [RegionInfo](#) members can differ depending on whether the [RegionInfo](#) object was instantiated by using a culture name or a region name. For example, the following table lists the differences in return values when a [RegionInfo](#) object is instantiated by using the "US" region, the "en-US" culture, and the "es-US" culture.

 Expand table

| Member | "US" | "en-US" | "es-US" |
|------------------------------------|---------------|---------------|-----------------|
| CurrencyNativeName | US Dollar | US Dollar | Dólar de EE.UU. |
| Name | US | en-US | es-US |
| NativeName | United States | United States | Estados Unidos |
| ToString | US | en-US | es-US |

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



.NET feedback

.NET is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

SortKey class

Article • 12/30/2023

This article provides supplementary remarks to the reference documentation for this API.

A culture-sensitive comparison of two strings depends on each character in the strings having several categories of sort weights, including script, alphabetic, case, and diacritic weights. A sort key serves as the repository of these weights for a particular string.

The [CompareInfo.GetSortKey](#) method returns an instance of the [SortKey](#) class that reflects the culture-sensitive mapping of characters in a specified string. The value of a [SortKey](#) object is its key data, which is returned by the [KeyData](#) property. This key data consists of a series of bytes that encode the string, culture-specific sorting rules, and user-specified comparison options. A comparison using sort keys consists of a bitwise comparison of the corresponding key data in each sort key. For example, if you create a sort key by calling the [GetSortKey\(String, CompareOptions\)](#) method with a value of [CompareOptions.IgnoreCase](#), a string comparison operation that uses the sort key is case-insensitive.

After you create a sort key for a string, you compare sort keys by calling the static [SortKey.Compare](#) method. This method performs a simple byte-by-byte comparison, so it is much faster than the [String.Compare](#) or [CompareInfo.Compare](#) method.

ⓘ Note

You can download the [Sorting Weight Tables](#), a set of text files that contain information on the character weights used in sorting and comparison operations for Windows operating systems, the [Default Unicode Collation Element Table](#), the sort weight table for Linux and macOS.

Performance considerations

When performing a string comparison, the [Compare](#) and [CompareInfo.Compare](#) methods yield the same results, but they target different scenarios.

At a high level, the [CompareInfo.Compare](#) method generates the sort key for each string, performs the comparison, and then discards the sort key and returns the result of the comparison. However, the [CompareInfo.Compare](#) method actually doesn't generate an entire sort key to perform the comparison. Instead, the method generates the key

data for each text element (that is, base character, surrogate pair, or combining character sequence) in each string. The method then compares the key data for the corresponding text elements. The operation terminates as soon as the ultimate result of the comparison is determined. Sort key information is computed, but no [SortKey](#) object is created. This strategy is economical in terms of performance if both strings are compared once, but becomes expensive if the same strings are compared many times.

The [Compare](#) method requires generation of a [SortKey](#) object for each string before performing the comparison. This strategy is expensive in terms of performance for the first comparison because of the time and memory invested to generate the [SortKey](#) objects. However, it becomes economical if the same sort keys are compared many times.

For example, suppose you write an application that searches a database table for the row in which the string-based index column matches a specified search string. The table contains thousands of rows, and comparing the search string to the index in each row will take a long time. Therefore, when the application stores a row and its index column, it also generates and stores the sort key for the index in a column dedicated to improving search performance. When the application searches for a target row, it compares the sort key for the search string to the sort key for the index string, instead of comparing the search string to the index string.

Security considerations

The [CompareInfo.GetSortKey\(String, CompareOptions\)](#) method returns a [SortKey](#) object with the value based on a specified string and [CompareOptions](#) value, and the culture associated with the underlying [CompareInfo](#) object. If a security decision depends on a string comparison or case change, you should use the [CompareInfo.GetSortKey\(String, CompareOptions\)](#) method of the invariant culture to ensure that the behavior of the operation is consistent, regardless of the culture settings of the operating system.

Use the following steps to obtain a sort key:

1. Retrieve the invariant culture from the [CultureInfo.InvariantCulture](#) property.
2. Retrieve a [CompareInfo](#) object for the invariant culture from the [CultureInfo.CompareInfo](#) property.
3. Call the [CompareInfo.GetSortKey\(String, CompareOptions\)](#) method.

Working with the value of a [SortKey](#) object is equivalent to calling the Windows [LCMapString](#) method with the LCMAP_SORTKEY value specified. However, for the

[SortKey](#) object, the sort keys for English characters precede the sort keys for Korean characters.

[SortKey](#) objects can be serialized, but only so that they can cross [AppDomain](#) objects. If an application serializes a [SortKey](#) object, the application must regenerate all the sort keys when there is a new version of the .NET Framework.

For more information about sort keys, see Unicode Technical Standard #10, "Unicode Collation Algorithm" on the [Unicode Consortium website](#).

 [Collaborate with us on GitHub](#)

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

 [.NET feedback](#)

.NET is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

SortVersion class

Article • 12/30/2023

This article provides supplementary remarks to the reference documentation for this API.

Sorting and string comparison in .NET Framework

Through .NET Framework 4, each version of .NET Framework included tables that contained sort weights and data on string normalization and that are based on a particular version of Unicode. In .NET Framework 4.5 and later versions, the presence of these tables depends on the operating system:

- On Windows 7 and previous versions, the tables continue to be used for comparing and ordering strings.
- On Windows 8, .NET Framework delegates string comparison and ordering operations to the operating system.

Consequently, the result of a string comparison can depend not only on the .NET Framework version, but also on the operating system version, as the following table shows. Note that this list of supported Unicode versions applies to character comparison and sorting only; it does not apply to classification of Unicode characters by category.

[+] Expand table

| .NET Framework version | Operating system | Unicode version |
|------------------------|------------------------------|-----------------|
| 4 | All operating systems | Unicode 5.0 |
| 4.5 and later versions | Windows 7 | Unicode 5.0 |
| 4.5 and later versions | Windows 8 and later versions | Unicode 6.0 |

On Windows 8, because the version of Unicode used in string comparison and ordering depends on the version of the operating system, the results of string comparison may differ even for applications that run on a specific version of .NET Framework.

Sorting and string comparison in .NET Core

All versions of .NET (Core) rely on the underlying operating system when performing string comparisons. Therefore, the results of a string comparison or the order in which strings are sorted depends on the version of Unicode used by the operating system when performing the comparison. On Linux, macOS, and Windows 10 and later versions, [International Components for Unicode](#) ↗ libraries provide the implementation for comparison and sorting APIs.

Use the `SortVersion` class

The `SortVersion` class provides information about the Unicode version used by .NET for string comparison and ordering. It enables developers to write applications that can detect and successfully handle changes in the version of Unicode that is used to compare and sort an application's strings.

You can instantiate a `SortVersion` object in two ways:

- By calling the `SortVersion` constructor, which instantiates a new `SortVersion` object based on a version number and sort ID. This constructor is most useful when recreating a `SortVersion` object from saved data.
- By retrieving the value of the `CompareInfo.Version` property. This property provides information about the Unicode version used by the .NET implementation on which the application is running.

The `SortVersion` class has two properties, `FullVersion` and `SortId`, that indicate the Unicode version and the specific culture used for string comparison. The `FullVersion` property is an arbitrary numeric value that reflects the Unicode version used for string comparison, and the `SortId` property is an arbitrary `Guid` that reflects the culture whose conventions are used for string comparison. The values of these two properties are important only when you compare two `SortVersion` objects by using the `Equals` method, the `Equality` operator, or the `Inequality` operator.

You typically use a `SortVersion` object when saving or retrieving some form of culture-sensitive, ordered string data, such as indexes or the literal strings themselves. This requires the following steps:

1. When the ordered string data is saved, the `FullVersion` and `SortId` property values are also saved.
2. When the ordered string data is retrieved, you can recreate the `SortVersion` object used for ordering the strings by calling the `SortVersion` constructor.
3. This newly instantiated `SortVersion` object is compared with a `SortVersion` object that reflects the culture whose conventions are used to order the string data.

4. If the two `SortVersion` objects are not equal, the string data must be reordered.

 **Collaborate with us on GitHub**

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

.NET is an open source project.
Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

Resources in .NET apps

Article • 03/18/2023

Nearly every production-quality app has to use resources. A resource is any non-executable data that is logically deployed with an app. A resource might be displayed in an app as error messages or as part of the user interface. Resources can contain data in a number of forms, including strings, images, and persisted objects. (To write persisted objects to a resource file, the objects must be serializable.) Storing your data in a resource file enables you to change the data without recompiling your entire app. It also enables you to store data in a single location, and eliminates the need to rely on hard-coded data that is stored in multiple locations.

.NET provides comprehensive support for the creation and [localization](#) of resources. In addition, .NET supports a simple model for packaging and deploying localized resources.

Create and localize resources

In a non-localized app, you can use resource files as a repository for app data, particularly for strings that might otherwise be hard-coded in multiple locations in source code. Most commonly, you create resources as either text (.txt) or XML (.resx) files, and use [Resgen.exe \(Resource File Generator\)](#) to compile them into binary .resources files. These files can then be embedded in the app's executable file by a language compiler. For more information about creating resources, see [Create resource files](#).

You can also localize your app's resources for specific cultures. This enables you to build localized (translated) versions of your apps. When you develop an app that uses localized resources, you designate a culture that serves as the neutral or fallback culture whose resources are used if no suitable resources are available. Typically, the resources of the neutral culture are stored in the app's executable. The remaining resources for individual localized cultures are stored in standalone satellite assemblies. For more information, see [Create satellite assemblies](#).

Package and deploy resources

You deploy localized app resources in [satellite assemblies](#). A satellite assembly contains the resources of a single culture; it does not contain any app code. In the satellite assembly deployment model, you create an app with one default assembly (which is typically the main assembly) and one satellite assembly for each culture that the app

supports. Because the satellite assemblies are not part of the main assembly, you can easily replace or update resources corresponding to a specific culture without replacing the app's main assembly.

Carefully determine which resources will make up your app's default resource assembly. Because it is a part of the main assembly, any changes to it will require you to replace the main assembly. If you do not provide a default resource, an exception will be thrown when the [resource fallback process](#) attempts to find it. In a well-designed app, using resources should never throw an exception.

For more information, see the [Packaging and Deploying Resources](#) article.

Retrieve resources

At run time, an app loads the appropriate localized resources on a per-thread basis, based on the culture specified by the [CultureInfo.CurrentCulture](#) property. This property value is derived as follows:

- By directly assigning a [CultureInfo](#) object that represents the localized culture to the [Thread.CurrentCulture](#) property.
- If a culture is not explicitly assigned, by retrieving the default thread UI culture from the [CultureInfo.DefaultThreadCurrentUICulture](#) property.
- If a default thread UI culture is not explicitly assigned, by retrieving the culture for the current user on the local computer. .NET implementations running on Windows do this by calling the Windows [GetUserDefaultUILanguage](#) function.

For more information about how the current UI culture is set, see the [CultureInfo](#) and [CultureInfo.CurrentCulture](#) reference pages.

You can then retrieve resources for the current UI culture or for a specific culture by using the [System.Resources.ResourceManager](#) class. Although the [ResourceManager](#) class is most commonly used for retrieving resources, the [System.Resources](#) namespace contains additional types that you can use to retrieve resources. These include:

- The [ResourceReader](#) class, which enables you to enumerate resources embedded in an assembly or stored in a standalone binary .resources file. It is useful when you don't know the precise names of the resources that are available at run time.
- The [ResXResourceReader](#) class, which enables you to retrieve resources from an XML (.resx) file.

- The [ResourceSet](#) class, which enables you to retrieve the resources of a specific culture without observing fallback rules. The resources can be stored in an assembly or a standalone binary .resources file. You can also develop an [IResourceReader](#) implementation that enables you to use the [ResourceSet](#) class to retrieve resources from some other source.
- The [ResXResourceSet](#) class, which enables you to retrieve all the items in an XML resource file into memory.

See also

- [CultureInfo](#)
- [CultureInfo.CurrentCulture](#)
- [Create resource files](#)
- [Package and deploy resources](#)
- [Create satellite assemblies](#)
- [Retrieve resources](#)
- [Localization in .NET](#)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

Create resource files for .NET apps

Article • 03/18/2023

You can include resources, such as strings, images, or object data, in resources files to make them easily available to your application. The .NET Framework offers five ways to create resources files:

- Create a text file that contains string resources. You can use [Resource File Generator \(resgen.exe\)](#) to convert the text file into a binary resource (.resources) file. You can then embed the binary resource file in an application executable or an application library by using a language compiler, or you can embed it in a satellite assembly by using [Assembly Linker \(Al.exe\)](#). For more information, see the [Resources in text files](#) section.
- Create an XML resource (.resx) file that contains string, image, or object data. You can use [Resource File Generator \(resgen.exe\)](#) to convert the .resx file into a binary resource (.resources) file. You can then embed the binary resource file in an application executable or an application library by using a language compiler, or you can embed it in a satellite assembly by using [Assembly Linker \(Al.exe\)](#). For more information, see the [Resources in .resx Files](#) section.
- Create an XML resource (.resx) file programmatically by using types in the [System.Resources](#) namespace. You can create a .resx file, enumerate its resources, and retrieve specific resources by name. For more information, see [Working with .resx Files Programmatically](#).
- Create a binary resource (.resources) file programmatically. You can then embed the file in an application executable or an application library by using a language compiler, or you can embed it in a satellite assembly by using [Assembly Linker \(Al.exe\)](#). For more information, see the [Resources in .resources Files](#) section.
- Use [Visual Studio](#) to create a resource file and include it in your project. Visual Studio provides a resource editor that lets you add, delete, and modify resources. At compile time, the resource file is automatically converted to a binary .resources file and embedded in an application assembly or satellite assembly. For more information, see the [Resource files in Visual Studio](#) section.

Resources in text files

You can use text (.txt or .restext) files to store string resources only. For non-string resources, use .resx files or create them programmatically. Text files that contain string

resources have the following format:

```
text

# This is an optional comment.
name = value

; This is another optional comment.
name = value

; The following supports conditional compilation if X is defined.
#ifndef X
name1=value1
name2=value2
#endif

# The following supports conditional compilation if Y is undefined.
#ifndef !Y
name1=value1
name2=value2
#endif
```

The resource file format of .txt and .restext files is identical. The .restext file extension merely serves to make text files immediately identifiable as text-based resource files.

String resources appear as *name/value* pairs, where *name* is a string that identifies the resource, and *value* is the resource string that is returned when you pass *name* to a resource retrieval method such as [ResourceManager.GetString](#). *name* and *value* must be separated by an equal sign (=). For example:

```
text

FileMenuName=File
EditMenuName>Edit
ViewMenuName=View
HelpMenuName=Help
```

⊗ Caution

Do not use resource files to store passwords, security-sensitive information, or private data.

Empty strings (that is, a resource whose value is [String.Empty](#)) are permitted in text files. For example:

```
text
```

```
EmptyString=
```

Starting with .NET Framework 4.5 and in all versions of .NET Core, text files support conditional compilation with the `#ifdef symbol... #endif` and `#if !symbol... #endif` constructs. You can then use the `/define` switch with [Resource File Generator \(resgen.exe\)](#) to define symbols. Each resource requires its own `#ifdef symbol... #endif` or `#if !symbol... #endif` construct. If you use an `#ifdef` statement and *symbol* is defined, the associated resource is included in the .resources file; otherwise, it is not included. If you use an `#if !` statement and *symbol* is not defined, the associated resource is included in the .resources file; otherwise, it is not included.

Comments are optional in text files and are preceded either by a semicolon (;) or by a pound sign (#) at the beginning of a line. Lines that contain comments can be placed anywhere in the file. Comments are not included in a compiled .resources file that is created by using [Resource File Generator \(resgen.exe\)](#).

Any blank lines in the text files are considered to be white space and are ignored.

The following example defines two string resources named `OKButton` and `CancelButton`.

```
text
```

```
#Define resources for buttons in the user interface.  
OKButton=OK  
CancelButton=Cancel
```

If the text file contains duplicate occurrences of *name*, [Resource File Generator \(resgen.exe\)](#) displays a warning and ignores the second name.

value cannot contain new line characters, but you can use C language-style escape characters such as `\n` to represent a new line and `\t` to represent a tab. You can also include a backslash character if it is escaped (for example, "`\`"). In addition, an empty string is permitted.

Save resources in text file format by using UTF-8 encoding or UTF-16 encoding in either little-endian or big-endian byte order. However, [Resource File Generator \(resgen.exe\)](#), which converts a .txt file to a .resources file, treats files as UTF-8 by default. If you want Resgen.exe to recognize a file that was encoded using UTF-16, you must include a Unicode byte order mark (U+FEFF) at the beginning of the file.

To embed a resource file in text format into a .NET assembly, you must convert the file to a binary resource (.resources) file by using [Resource File Generator \(resgen.exe\)](#). You

can then embed the .resources file in a .NET assembly by using a language compiler or embed it in a satellite assembly by using [Assembly Linker \(Al.exe\)](#).

The following example uses a resource file in text format named GreetingResources.txt for a simple "Hello World" console application. The text file defines two strings, `prompt` and `greeting`, that prompt the user to enter their name and display a greeting.

text

```
# GreetingResources.txt
# A resource file in text format for a "Hello World" application.
#
# Initial prompt to the user.
prompt=Enter your name:
# Format string to display the result.
greeting=Hello, {0}!
```

The text file is converted to a .resources file by using the following command:

Console

```
resgen GreetingResources.txt
```

The following example shows the source code for a console application that uses the .resources file to display messages to the user.

C#

```
using System;
using System.Reflection;
using System.Resources;

public class Example
{
    public static void Main()
    {
        ResourceManager rm = new ResourceManager("GreetingResources",
                                                typeof(Example).Assembly);
        Console.WriteLine(rm.GetString("prompt"));
        string name = Console.ReadLine();
        Console.WriteLine(rm.GetString("greeting"), name);
    }
}
// The example displays output like the following:
//      Enter your name: Wilberforce
//      Hello, Wilberforce!
```

If you are using Visual Basic, and the source code file is named Greeting.vb, the following command creates an executable file that includes the embedded .resources file:

```
Console  
vbc greeting.vb -resource:GreetingResources.resources
```

If you are using C#, and the source code file is named Greeting.cs, the following command creates an executable file that includes the embedded .resources file:

```
Console  
csc greeting.cs -resource:GreetingResources.resources
```

Resources in .resx files

Unlike text files, which can only store string resources, XML resource (.resx) files can store strings, binary data such as images, icons, and audio clips, and programmatic objects. A .resx file contains a standard header, which describes the format of the resource entries and specifies the versioning information for the XML that is used to parse the data. The resource file data follows the XML header. Each data item consists of a name/value pair that is contained in a `data` tag. Its `name` attribute defines the resource name, and the nested `value` tag contains the resource value. For string data, the `value` tag contains the string.

For example, the following `data` tag defines a string resource named `prompt` whose value is "Enter your name:".

```
XML  
<data name="prompt" xml:space="preserve">  
  <value>Enter your name:</value>  
</data>
```

⚠ Warning

Do not use resource files to store passwords, security-sensitive information, or private data.

For resource objects, the `data` tag includes a `type` attribute that indicates the data type of the resource. For objects that consist of binary data, the `data` tag also includes a `mimetype` attribute, which indicates the `base64` type of the binary data.

Note

All .resx files use a binary serialization formatter to generate and parse the binary data for a specified type. As a result, a .resx file can become invalid if the binary serialization format for an object changes in an incompatible way.

The following example shows a portion of a .resx file that includes an `Int32` resource and a bitmap image.

XML

```
<data name="i1" type="System.Int32, mscorelbin">
  <value>20</value>
</data>

<data name="flag" type="System.Drawing.Bitmap, System.Drawing,
  Version=1.0.5000.0, Culture=neutral, PublicKeyToken=b03f5f7f11d50a3a"
  mimetype="application/x-microsoft.net.object.bytearray.base64">
  <value>
    AAEAAAD/////AQAAAAAAAAMAgAAADtTeX...
  </value>
</data>
```

Important

Because .resx files must consist of well-formed XML in a predefined format, we do not recommend working with .resx files manually, particularly when the .resx files contain resources other than strings. Instead, [Visual Studio](#) provides a transparent interface for creating and manipulating .resx files. For more information, see the [Resource files in Visual Studio](#) section. You can also create and manipulate .resx files programmatically. For more information, see [Work with .resx files programmatically](#).

Resources in .resources files

You can use the [System.Resources.ResourceWriter](#) class to programmatically create a binary resource (.resources) file directly from code. You can also use [Resource File Generator \(resgen.exe\)](#) to create a .resources file from a text file or a .resx file. The

.resources file can contain binary data (byte arrays) and object data in addition to string data. Programmatically creating a .resources file requires the following steps:

1. Create a [ResourceWriter](#) object with a unique file name. You can do this by specifying either a file name or a file stream to a [ResourceWriter](#) class constructor.
2. Call one of the overloads of the [ResourceWriter.AddResource](#) method for each named resource to add to the file. The resource can be a string, an object, or a collection of binary data (a byte array).
3. Call the [ResourceWriter.Close](#) method to write the resources to the file and to close the [ResourceWriter](#) object.

 **Note**

Do not use resource files to store passwords, security-sensitive information, or private data.

The following example programmatically creates a .resources file named CarResources.resources that stores six strings, an icon, and two application-defined objects (two `Automobile` objects). The `Automobile` class, which is defined and instantiated in the example, is tagged with the [SerializableAttribute](#) attribute, which allows it to be persisted by the binary serialization formatter.

C#

```
using System;
using System.Drawing;
using System.Resources;

[Serializable()] public class Automobile
{
    private string carMake;
    private string carModel;
    private int carYear;
    private int carDoors;
    private int carCylinders;

    public Automobile(string make, string model, int year) :
        this(make, model, year, 0, 0)
    { }

    public Automobile(string make, string model, int year,
                      int doors, int cylinders)
    {
        this.carMake = make;
        this.carModel = model;
```

```
        this.carYear = year;
        this.carDoors = doors;
        this.carCylinders = cylinders;
    }

    public string Make {
        get { return this.carMake; }
    }

    public string Model {
        get { return this.carModel; }
    }

    public int Year {
        get { return this.carYear; }
    }

    public int Doors {
        get {
            return this.carDoors; }
    }

    public int Cylinders {
        get {
            return this.carCylinders; }
    }
}

public class Example
{
    public static void Main()
    {
        // Instantiate an Automobile object.
        Automobile car1 = new Automobile("Ford", "Model N", 1906, 0, 4);
        Automobile car2 = new Automobile("Ford", "Model T", 1909, 2, 4);
        // Define a resource file named CarResources.resx.
        using (ResourceWriter rw = new
ResourceWriter(@".\CarResources.resources"))
        {
            rw.AddResource("Title", "Classic American Cars");
            rw.AddResource("HeaderString1", "Make");
            rw.AddResource("HeaderString2", "Model");
            rw.AddResource("HeaderString3", "Year");
            rw.AddResource("HeaderString4", "Doors");
            rw.AddResource("HeaderString5", "Cylinders");
            rw.AddResource("Information", SystemIcons.Information);
            rw.AddResource("EarlyAuto1", car1);
            rw.AddResource("EarlyAuto2", car2);
        }
    }
}
```

After you create the .resources file, you can embed it in a run-time executable or library by including the language compiler's `/resource` switch, or embed it in a satellite assembly by using [Assembly Linker \(Al.exe\)](#).

Resource files in Visual Studio

When you add a resource file to your [Visual Studio](#) project, Visual Studio creates a .resx file in the project directory. Visual Studio provides resource editors that enable you to add strings, images, and binary objects. Because the editors are designed to handle static data only, they cannot be used to store programmatic objects; you must write object data to either a .resx file or to a .resources file programmatically. For more information, see [Work with .resx files programmatically](#) and the [Resources in .resources files](#) section.

If you're adding localized resources, give them the same root file name as the main resource file. You should also designate their culture in the file name. For example, if you add a resource file named *Resources.resx*, you might also create resource files named *Resources.en-US.resx* and *Resources.fr-FR.resx* to hold localized resources for the English (United States) and French (France) cultures, respectively. You should also designate your application's default culture. This is the culture whose resources are used if no localized resources for a particular culture can be found.

To specify the default culture, in **Solution Explorer** in Visual Studio:

- Open the project properties, right-click the project and select **Properties** (or **Alt** + **Enter** when project is selected).
- Select the **Package** tab.
- In the **General** area, select the appropriate language/culture from the **Assembly neutral language** control.
- Save your changes.

At compile time, Visual Studio first converts the .resx files in a project to binary resource (.resources) files and stores them in a subdirectory of the project's *obj* directory. Visual Studio embeds any resource files that do not contain localized resources in the main assembly that is generated by the project. If any resource files contain localized resources, Visual Studio embeds them in separate satellite assemblies for each localized culture. It then stores each satellite assembly in a directory whose name corresponds to the localized culture. For example, localized English (United States) resources are stored in a satellite assembly in the en-US subdirectory.

See also

- [System.Resources](#)
- [Resources in .NET Apps](#)
- [Package and deploy resources](#)

 [Collaborate with us on GitHub](#)

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

 .NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

Work with .resx files programmatically

Article • 03/18/2023

ⓘ Note

This article applies to .NET Framework. For information that applies to .NET 5+ (including .NET Core), see [Resources in .resx files](#).

Because XML resource (.resx) files must consist of well-defined XML, including a header that must follow a specific schema followed by data in name/value pairs, you may find that creating these files manually is error-prone. As an alternative, you can create .resx files programmatically by using types and members in the .NET Class Library. You can also use the .NET Class Library to retrieve resources that are stored in .resx files. This article explains how you can use the types and members in the [System.Resources](#) namespace to work with .resx files.

This article discusses working with XML (.resx) files that contain resources. For information on working with binary resource files that have been embedded in assemblies, see [ResourceManager](#).

⚠ Warning

There are also ways to work with .resx files other than programmatically. When you add a resource file to a [Visual Studio](#) project, Visual Studio provides an interface for creating and maintaining a .resx file, and automatically converts the .resx file to a .resources file at compile time. You can also use a text editor to manipulate a .resx file directly. However, to avoid corrupting the file, be careful not to modify any binary information that is stored in the file.

Create a .resx file

You can use the [System.Resources.ResXResourceWriter](#) class to create a .resx file programmatically, by following these steps:

1. Instantiate a [ResXResourceWriter](#) object by calling the [ResXResourceWriter\(String\)](#) method and supplying the name of the .resx file. The file name must include the .resx extension. If you instantiate the [ResXResourceWriter](#) object in a `using` block, you do not explicitly have to call the [ResXResourceWriter.Close](#) method in step 3.

2. Call the [ResXResourceWriter.AddResource](#) method for each resource you want to add to the file. Use the overloads of this method to add string, object, and binary (byte array) data. If the resource is an object, it must be serializable.
3. Call the [ResXResourceWriter.Close](#) method to generate the resource file and to release all resources. If the [ResXResourceWriter](#) object was created within a `using` block, resources are written to the .resx file and the resources used by the [ResXResourceWriter](#) object are released at the end of the `using` block.

The resulting .resx file has the appropriate header and a `data` tag for each resource added by the [ResXResourceWriter.AddResource](#) method.

 **Warning**

Do not use resource files to store passwords, security-sensitive information, or private data.

The following example creates a .resx file named CarResources.resx that stores six strings, an icon, and two application-defined objects (two `Automobile` objects). The `Automobile` class, which is defined and instantiated in the example, is tagged with the [SerializableAttribute](#) attribute.

C#

```
using System;
using System.Drawing;
using System.Resources;

[Serializable()] public class Automobile
{
    private string carMake;
    private string carModel;
    private int carYear;
    private int carDoors;
    private int carCylinders;

    public Automobile(string make, string model, int year) :
        this(make, model, year, 0, 0)
    { }

    public Automobile(string make, string model, int year,
                      int doors, int cylinders)
    {
        this.carMake = make;
        this.carModel = model;
        this.carYear = year;
        this.carDoors = doors;
    }
}
```

```

        this.carCylinders = cylinders;
    }

    public string Make {
        get { return this.carMake; }
    }

    public string Model {
        get { return this.carModel; }
    }

    public int Year {
        get { return this.carYear; }
    }

    public int Doors {
        get { return this.carDoors; }
    }

    public int Cylinders {
        get { return this.carCylinders; }
    }
}

public class Example
{
    public static void Main()
    {
        // Instantiate an Automobile object.
        Automobile car1 = new Automobile("Ford", "Model N", 1906, 0, 4);
        Automobile car2 = new Automobile("Ford", "Model T", 1909, 2, 4);
        // Define a resource file named CarResources.resx.
        using (ResXResourceWriter resx = new
ResXResourceWriter(@".\CarResources.resx"))
        {
            resx.AddResource("Title", "Classic American Cars");
            resx.AddResource("HeaderString1", "Make");
            resx.AddResource("HeaderString2", "Model");
            resx.AddResource("HeaderString3", "Year");
            resx.AddResource("HeaderString4", "Doors");
            resx.AddResource("HeaderString5", "Cylinders");
            resx.AddResource("Information", SystemIcons.Information);
            resx.AddResource("EarlyAuto1", car1);
            resx.AddResource("EarlyAuto2", car2);
        }
    }
}

```

💡 Tip

You can also use **Visual Studio** to create .resx files. At compile time, Visual Studio uses the **Resource File Generator (Resgen.exe)** to convert the .resx file to a binary

resource (.resources) file, and also embeds it in either an application assembly or a satellite assembly.

You cannot embed a .resx file in a runtime executable or compile it into a satellite assembly. You must convert your .resx file into a binary resource (.resources) file by using the [Resource File Generator \(Resgen.exe\)](#). The resulting .resources file can then be embedded in an application assembly or a satellite assembly. For more information, see [Create resource files](#).

Enumerate resources

In some cases, you may want to retrieve all resources, instead of a specific resource, from a .resx file. To do this, you can use the [System.Resources.ResXResourceReader](#) class, which provides an enumerator for all resources in the .resx file. The [System.Resources.ResXResourceReader](#) class implements [IDictionaryEnumerator](#), which returns a [DictionaryEntry](#) object that represents a particular resource for each iteration of the loop. Its [DictionaryEntry.Key](#) property returns the resource's key, and its [DictionaryEntry.Value](#) property returns the resource's value.

The following example creates a [ResXResourceReader](#) object for the CarResources.resx file created in the previous example and iterates through the resource file. It adds the two `Automobile` objects that are defined in the resource file to a [System.Collections.Generic.List<T>](#) object, and it adds five of the six strings to a [SortedList](#) object. The values in the [SortedList](#) object are converted to a parameter array, which is used to display column headings to the console. The `Automobile` property values are also displayed to the console.

C#

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Resources;

public class Example
{
    public static void Main()
    {
        string resxFile = @"\CarResources.resx";
        List<Automobile> autos = new List<Automobile>();
        SortedList headers = new SortedList();

        using (ResXResourceReader resxReader = new
ResXResourceReader(resxFile))
        {
```

```

        foreach (DictionaryEntry entry in resxReader) {
            if (((string) entry.Key).StartsWith("EarlyAuto"))
                autos.Add((Automobile) entry.Value);
            else if (((string) entry.Key).StartsWith("Header"))
                headers.Add((string) entry.Key, (string) entry.Value);
        }
    }
    string[] headerColumns = new string[headers.Count];
    headers.GetValueList().CopyTo(headerColumns, 0);
    Console.WriteLine("{0,-8} {1,-10} {2,-4} {3,-5} {4,-9}\n",
                      headerColumns);
    foreach (var auto in autos)
        Console.WriteLine("{0,-8} {1,-10} {2,4} {3,5} {4,9}",
                         auto.Make, auto.Model, auto.Year,
                         auto.Doors, auto.Cylinders);
}
// The example displays the following output:
//      Make      Model      Year   Doors   Cylinders
//
//      Ford      Model N    1906      0       4
//      Ford      Model T    1909      2       4

```

Retrieve a specific resource

In addition to enumerating the items in a .resx file, you can retrieve a specific resource by name by using the [System.Resources.ResXResourceSet](#) class. The [ResourceSet.GetString\(String\)](#) method retrieves the value of a named string resource. The [ResourceSet.GetObject\(String\)](#) method retrieves the value of a named object or binary data. The method returns an object that must then be cast (in C#) or converted (in Visual Basic) to an object of the appropriate type.

The following example retrieves a form's caption string and icon by their resource names. It also retrieves the application-defined `Automobile` objects used in the previous example and displays them in a [DataGridView](#) control.

C#

```

using System;
using System.Collections.Generic;
using System.Drawing;
using System.Resources;
using System.Windows.Forms;

public class CarDisplayApp : Form
{
    private const string resxFile = @"\CarResources.resx";
    Automobile[] cars;

```

```

public static void Main()
{
    CarDisplayApp app = new CarDisplayApp();
    Application.Run(app);
}

public CarDisplayApp()
{
    // Instantiate controls.
    PictureBox pictureBox = new PictureBox();
    pictureBox.Location = new Point(10, 10);
    this.Controls.Add(pictureBox);
    DataGridView grid = new DataGridView();
    grid.Location = new Point(10, 60);
    this.Controls.Add(grid);

    // Get resources from .resx file.
    using (ResXResourceSet resxSet = new ResXResourceSet(resxFfile))
    {
        // Retrieve the string resource for the title.
        this.Text = resxSet.GetString("Title");
        // Retrieve the image.
        Icon image = (Icon) resxSet.GetObject("Information", true);
        if (image != null)
            pictureBox.Image = image.ToBitmap();

        // Retrieve Automobile objects.
        List<Automobile> carList = new List<Automobile>();
        string resName = "EarlyAuto";
        Automobile auto;
        int ctr = 1;
        do {
            auto = (Automobile) resxSet.GetObject(resName + ctr.ToString());
            ctr++;
            if (auto != null)
                carList.Add(auto);
        } while (auto != null);
        cars = carList.ToArray();
        grid.DataSource = cars;
    }
}
}

```

Convert .resx files to binary .resources files

Converting .resx files to embedded binary resource (.resources) files has significant advantages. Although .resx files are easy to read and maintain during application development, they are rarely included with finished applications. If they are distributed with an application, they exist as separate files apart from the application executable and its accompanying libraries. In contrast, .resources files are embedded in the

application executable or its accompanying assemblies. In addition, for localized applications, relying on .resx files at run time places the responsibility for handling resource fallback on the developer. In contrast, if a set of satellite assemblies that contain embedded .resources files has been created, the common language runtime handles the resource fallback process.

To convert a .resx file to a .resources file, you use [Resource File Generator \(resgen.exe\)](#), which has the following basic syntax:

Console

```
resgen.exe .resxFilename
```

The result is a binary resource file that has the same root file name as the .resx file and a .resources file extension. This file can then be compiled into an executable or a library at compile time. If you are using the Visual Basic compiler, use the following syntax to embed a .resources file in an application's executable:

Console

```
vbc filename .vb -resource: .resourcesFilename
```

If you are using C#, the syntax is as follows:

Console

```
csc filename .cs -resource: .resourcesFilename
```

The .resources file can also be embedded in a satellite assembly by using [Assembly Linker \(al.exe\)](#), which has the following basic syntax:

Console

```
al resourcesFilename -out: assemblyFilename
```

See also

- [Create resource files](#)
- [Resource File Generator \(resgen.exe\)](#)
- [Assembly Linker \(al.exe\)](#)

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

Create satellite assemblies for .NET apps

Article • 03/18/2023

Resource files play a central role in localized applications. They enable an application to display strings, images, and other data in the user's language and culture, and provide alternate data if resources for the user's language or culture are unavailable. .NET uses a hub-and-spoke model to locate and retrieve localized resources. The hub is the main assembly that contains the non-localizable executable code and the resources for a single culture, which is called the neutral or default culture. The default culture is the fallback culture for the application; it's used when no localized resources are available. You use the [NeutralResourcesLanguageAttribute](#) attribute to designate the culture of the application's default culture. Each spoke connects to a satellite assembly that contains the resources for a single localized culture but does not contain any code. Because the satellite assemblies aren't part of the main assembly, you can easily update or replace resources that correspond to a specific culture without replacing the main assembly for the application.

Note

The resources of an application's default culture can also be stored in a satellite assembly. To do this, you assign the [NeutralResourcesLanguageAttribute](#) attribute a value of [UltimateResourceFallbackLocation.Satellite](#).

Satellite assembly name and location

The hub-and-spoke model requires that you place resources in specific locations so that they can be easily located and used. If you don't compile and name resources as expected, or if you don't place them in the correct locations, the common language runtime won't be able to locate them and will use the resources of the default culture instead. The .NET resource manager is represented by the [ResourceManager](#) type, and it's used to automatically access localized resources. The resource manager requires the following:

- A single satellite assembly must include all the resources for a particular culture. In other words, you should compile multiple `.txt` or `.resx` files into a single binary `.resources` file.
- There must be a separate subdirectory in the application directory for each localized culture that stores that culture's resources. The subdirectory name must

be the same as the culture name. Alternately, you can store your satellite assemblies in the global assembly cache. In this case, the culture information component of the assembly's strong name must indicate its culture. For more information, see [Install satellite assemblies in the Global Assembly Cache](#).

 **Note**

If your application includes resources for subcultures, place each subculture in a separate subdirectory under the application directory. Do not place subcultures in subdirectories under their main culture's directory.

- The satellite assembly must have the same name as the application, and must use the file name extension ".resources.dll". For example, if an application is named *Example.exe*, the name of each satellite assembly should be *Example.resources.dll*. The satellite assembly name doesn't indicate the culture of its resource files. However, the satellite assembly appears in a directory that does specify the culture.
- Information about the culture of the satellite assembly must be included in the assembly's metadata. To store the culture name in the satellite assembly's metadata, you specify the `/culture` option when you use [Assembly Linker](#) to embed resources in the satellite assembly.

The following illustration shows a sample directory structure and location requirements for applications that you aren't installing in the [global assembly cache](#). The items with *.txt* and *.resources* extensions won't ship with the final application. These are the intermediate resource files used to create the final satellite resource assemblies. In this example, you could substitute *.resx* files for the *.txt* files. For more information, see [Package and deploy resources](#).

The following image shows the satellite assembly directory:

| | |
|----------------------|---|
| MyDir | Main directory for your application |
| MyApp.exe | Main assembly file. Strings.resources is embedded |
| strings.txt | File used to construct strings.resources |
| strings.resources | Default resources file |
| de | German subdirectory, where the German satellite is stored |
| strings.de.txt | Strings file, localized for German |
| strings.de.resources | Resources file, created from the strings file |
| MyApp.resources.dll | Satellite assembly, compiled from strings.de.resources |
| ja | Japanese subdirectory, where the Japanese satellite is stored |
| strings.ja.txt | Strings file, localized for Japanese |
| strings.ja.resources | Resources file, created from the strings file |
| MyApp.resources.dll | Satellite assembly, compiled from strings.ja.resources |
| ... | Additional subdirectories for each culture to support |

Compile satellite assemblies

You use [Resource File Generator \(*resgen.exe*\)](#) to compile text files or XML (.resx) files that contain resources to binary .resources files. You then use [Assembly Linker \(*al.exe*\)](#) to compile .resources files into satellite assemblies. *al.exe* creates an assembly from the .resources files that you specify. Satellite assemblies can contain only resources; they can't contain any executable code.

The following *al.exe* command creates a satellite assembly for the application [Example](#) from the German resources file *strings.de.resources*.

Console

```
al -target:lib -embed:strings.de.resources -culture:de -
out:Example.resources.dll
```

The following *al.exe* command also creates a satellite assembly for the application [Example](#) from the file *strings.de.resources*. The */template* option causes the satellite assembly to inherit all assembly metadata except for its culture information from the parent assembly (*Example.dll*).

Console

```
al -target:lib -embed:strings.de.resources -culture:de -
out:Example.resources.dll -template:Example.dll
```

The following table describes the *al.exe* options used in these commands in more detail:

| Option | Description |
|--|---|
| <code>-target:lib</code> | Specifies that your satellite assembly is compiled to a library (.dll) file. Because a satellite assembly doesn't contain executable code and is not an application's main assembly, you must save satellite assemblies as DLLs. |
| <code>-embed:strings.de.resources</code> | Specifies the name of the resource file to embed when <i>al.exe</i> compiles the assembly. You can embed multiple .resources files in a satellite assembly, but if you are following the hub-and-spoke model, you must compile one satellite assembly for each culture. However, you can create separate .resources files for strings and objects. |
| <code>-culture:de</code> | Specifies the culture of the resource to compile. The common language runtime uses this information when it searches for the resources for a specified culture. If you omit this option, <i>al.exe</i> will still compile the resource, but the runtime won't be able to find it when a user requests it. |
| <code>-out:Example.resources.dll</code> | Specifies the name of the output file. The name must follow the naming standard <i>baseName.resources.extension</i> , where <i>baseName</i> is the name of the main assembly and <i>extension</i> is a valid file name extension (such as .dll). The runtime is not able to determine the culture of a satellite assembly based on its output file name; you must use the <code>/culture</code> option to specify it. |
| <code>-template:Example.dll</code> | Specifies an assembly from which the satellite assembly will inherit all assembly metadata except the culture field. This option affects satellite assemblies only if you specify an assembly that has a strong name . |

For a complete list of options available with *al.exe*, see [Assembly Linker \(*al.exe*\)](#).

ⓘ Note

There may be times when you want to use the .NET Core MSBuild task to compile satellite assemblies, even though you're targeting .NET Framework. For example, you may want to use the C# compiler **deterministic** option to be able to compare assemblies from different builds. In this case, set `GenerateSatelliteAssembliesForCore` to `true` in the `.csproj` file to generate satellite assemblies using `csc.exe` instead of `Al.exe (Assembly Linker)`.

XML

```
<Project>
  <PropertyGroup>
```

```
<GenerateSatelliteAssembliesForCore>true</GenerateSatelliteAssembliesFo  
rCore>  
  </PropertyGroup>  
</Project>
```

The .NET Core MSBuild task uses `csc.exe` instead of `al.exe` to generate satellite assemblies, by default. For more information, see [Make it easier to opt into "Core" satellite assembly generation](#).

Satellite assemblies example

The following is a simple "Hello world" example that displays a message box containing a localized greeting. The example includes resources for the English (United States), French (France), and Russian (Russia) cultures, and its fallback culture is English. To create the example, do the following:

1. Create a resource file named `Greeting.resx` or `Greeting.txt` to contain the resource for the default culture. Store a single string named `HelloString` whose value is "Hello world!" in this file.
2. To indicate that English (en) is the application's default culture, add the following `System.Resources.NeutralResourcesLanguageAttribute` attribute to the application's `AssemblyInfo` file or to the main source code file that will be compiled into the application's main assembly.

C#

```
[assembly: NeutralResourcesLanguage("en")]
```

3. Add support for additional cultures (`en-US`, `fr-FR`, and `ru-RU`) to the application as follows:

- To support the `en-US` or English (United States) culture, create a resource file named `Greeting.en-US.resx` or `Greeting.en-US.txt`, and store in it a single string named `HelloString` whose value is "Hi world!".
- To support the `fr-FR` or French (France) culture, create a resource file named `Greeting.fr-FR.resx` or `Greeting.fr-FR.txt`, and store in it a single string named `HelloString` whose value is "Salut tout le monde!".
- To support the `ru-RU` or Russian (Russia) culture, create a resource file named `Greeting.ru-RU.resx` or `Greeting.ru-RU.txt`, and store in it a single string named

`HelloString` whose value is "Всем привет!".

4. Use `resgen.exe` to compile each text or XML resource file to a binary `.resources` file. The output is a set of files that have the same root file name as the `.resx` or `.txt` files, but a `.resources` extension. If you create the example with Visual Studio, the compilation process is handled automatically. If you aren't using Visual Studio, run the following commands to compile the `.resx` files into `.resources` files:

Console

```
resgen Greeting.resx
resgen Greeting.en-us.resx
resgen Greeting.fr-FR.resx
resgen Greeting.ru-RU.resx
```

If your resources are in text files instead of XML files, replace the `.resx` extension with `.txt`.

5. Compile the following source code along with the resources for the default culture into the application's main assembly:

 **Important**

If you are using the command line rather than Visual Studio to create the example, you should modify the call to the `ResourceManager` class constructor to the following: `ResourceManager rm = new ResourceManager("Greeting", typeof(Example).Assembly);`

C#

```
using System;
using System.Globalization;
using System.Reflection;
using System.Resources;
using System.Threading;
using System.Windows.Forms;

class Example
{
    static void Main()
    {
        // Create array of supported cultures
        string[] cultures = {"en-CA", "en-US", "fr-FR", "ru-RU"};
        Random rnd = new Random();
        int cultureNdx = rnd.Next(0, cultures.Length);
        CultureInfo originalCulture =
```

```
Thread.CurrentCulture;

try {
    CultureInfo newCulture = new
CultureInfo(cultures[cultureNdx]);
    Thread.CurrentCulture.CurrentCulture = newCulture;
    Thread.CurrentCulture.CurrentUICulture = newCulture;
    ResourceManager rm = new ResourceManager("Example.Greeting",
typeof(Example).Assembly);
    string greeting = String.Format("The current culture is
{0}.\n{1}",

Thread.CurrentCulture.CurrentCulture.Name,
rm.GetString("HelloString"));

    MessageBox.Show(greeting);
}
catch (CultureNotFoundException e) {
    Console.WriteLine("Unable to instantiate culture {0}",
e.InvalidCultureName);
}
finally {
    Thread.CurrentCulture.CurrentCulture = originalCulture;
    Thread.CurrentCulture.CurrentUICulture = originalCulture;
}
}
```

If the application is named **Example** and you're compiling from the command line, the command for the C# compiler is:

Console

The corresponding Visual Basic compiler command is:

Console

6. Create a subdirectory in the main application directory for each localized culture supported by the application. You should create an *en-US*, an *fr-FR*, and an *ru-RU* subdirectory. Visual Studio creates these subdirectories automatically as part of the compilation process.
 7. Embed the individual culture-specific *.resources* files into satellite assemblies and save them to the appropriate directory. The command to do this for each

.resources file is:

Console

```
al -target:lib -embed:Greeting.culture.resources -culture:culture -  
out:culture\Example.resources.dll
```

where *culture* is the name of the culture whose resources the satellite assembly contains. Visual Studio handles this process automatically.

You can then run the example. It will randomly make one of the supported cultures the current culture and display a localized greeting.

Install satellite assemblies in the Global Assembly Cache

Instead of installing assemblies in a local application subdirectory, you can install them in the global assembly cache. This is particularly useful if you have class libraries and class library resource assemblies that are used by multiple applications.

Installing assemblies in the global assembly cache requires that they have strong names. Strong-named assemblies are signed with a valid public/private key pair. They contain version information that the runtime uses to determine which assembly to use to satisfy a binding request. For more information about strong names and versioning, see [Assembly versioning](#). For more information about strong names, see [Strong-named assemblies](#).

When you're developing an application, it's unlikely that you'll have access to the final public/private key pair. To install a satellite assembly in the global assembly cache and ensure that it works as expected, you can use a technique called delayed signing. When you delay sign an assembly, at build time you reserve space in the file for the strong name signature. The actual signing is delayed until later, when the final public/private key pair is available. For more information about delayed signing, see [Delay signing an assembly](#).

Obtain the public key

To delay sign an assembly, you must have access to the public key. You can either obtain the real public key from the organization in your company that will do the eventual signing, or create a public key by using the [Strong Name tool \(sn.exe\)](#).

The following *Sn.exe* command creates a test public/private key pair. The **-k** option specifies that *Sn.exe* should create a new key pair and save it in a file named *TestKeyPair.snk*.

Console

```
sn -k TestKeyPair.snk
```

You can extract the public key from the file that contains the test key pair. The following command extracts the public key from *TestKeyPair.snk* and saves it in *PublicKey.snk*:

Console

```
sn -p TestKeyPair.snk PublicKey.snk
```

Delay signing an Assembly

After you obtain or create the public key, you use the [Assembly Linker \(*al.exe*\)](#) to compile the assembly and specify delayed signing.

The following *al.exe* command creates a strong-named satellite assembly for the application StringLibrary from the *strings.ja.resources* file:

Console

```
al -target:lib -embed:strings.ja.resources -culture:ja -  
out:StringLibrary.resources.dll -delay+ -keyfile:PublicKey.snk
```

The **-delay+** option specifies that the Assembly Linker should delay sign the assembly. The **-keyfile** option specifies the name of the key file that contains the public key to use to delay sign the assembly.

Re-signing an Assembly

Before you deploy your application, you must re-sign the delay signed satellite assembly with the real key pair. You can do this by using *Sn.exe*.

The following *Sn.exe* command signs *StringLibrary.resources.dll* with the key pair stored in the file *RealKeyPair.snk*. The **-R** option specifies that a previously signed or delay signed assembly is to be re-signed.

Console

```
sn -R StringLibrary.resources.dll RealKeyValuePair.snk
```

Install a satellite assembly in the Global Assembly Cache

When the runtime searches for resources in the resource fallback process, it looks in the [global assembly cache](#) first. (For more information, see the "Resource Fallback Process" section of [Package and deploy resources](#).) As soon as a satellite assembly is signed with a strong name, it can be installed in the global assembly cache by using the [Global Assembly Cache tool \(gacutil.exe\)](#).

The following *Gacutil.exe* command installs *StringLibrary.resources.dll** in the global assembly cache:

Console

```
gacutil -i:StringLibrary.resources.dll
```

The */i* option specifies that *Gacutil.exe* should install the specified assembly into the global assembly cache. After the satellite assembly is installed in the cache, the resources it contains become available to all applications that are designed to use the satellite assembly.

Resources in the Global Assembly Cache: An Example

The following example uses a method in a .NET class library to extract and return a localized greeting from a resource file. The library and its resources are registered in the global assembly cache. The example includes resources for the English (United States), French (France), Russian (Russia), and English cultures. English is the default culture; its resources are stored in the main assembly. The example initially delay-signs the library and its satellite assemblies with a public key, then re-signs them with a public/private key pair. To create the example, do the following:

1. If you aren't using Visual Studio, use the following [Strong Name Tool \(Sn.exe\)](#) command to create a public/private key pair named *ResKey.snk*:

Console

```
sn -k ResKey.snk
```

If you're using Visual Studio, use the **Signing** tab of the project **Properties** dialog box to generate the key file.

2. Use the following [Strong Name Tool \(Sn.exe\)](#) command to create a public key file named `PublicKey.snk`:

```
Console
```

```
sn -p ResKey.snk PublicKey.snk
```

3. Create a resource file named `Strings.resx` to contain the resource for the default culture. Store a single string named `Greeting` whose value is "How do you do?" in that file.
4. To indicate that "en" is the application's default culture, add the following [System.Resources.NeutralResourcesLanguageAttribute](#) attribute to the application's `AssemblyInfo` file or to the main source code file that will be compiled into the application's main assembly:

```
C#
```

```
[assembly:NeutralResourcesLanguageAttribute("en")]
```

5. Add support for additional cultures (the en-US, fr-FR, and ru-RU cultures) to the application as follows:

- To support the "en-US" or English (United States) culture, create a resource file named `Strings.en-US.resx` or `Strings.en-US.txt`, and store in it a single string named `Greeting` whose value is "Hello!".
- To support the "fr-FR" or French (France) culture, create a resource file named `Strings.fr-FR.resx` or `Strings.fr-FR.txt` and store in it a single string named `Greeting` whose value is "Bon jour!".
- To support the "ru-RU" or Russian (Russia) culture, create a resource file named `Strings.ru-RU.resx` or `Strings.ru-RU.txt` and store in it a single string named `Greeting` whose value is "Привет!".

6. Use [resgen.exe](#) to compile each text or XML resource file to a binary .resources file. The output is a set of files that have the same root file name as the .resx or .txt files, but a .resources extension. If you create the example with Visual Studio, the compilation process is handled automatically. If you aren't using Visual Studio, run the following command to compile the .resx files into .resources files:

```
Console
```

```
resgen filename
```

Where *filename* is the optional path, file name, and extension of the .resx or text file.

7. Compile the following source code for *StringLibrary.vb* or *StringLibrary.cs* along with the resources for the default culture into a delay signed library assembly named *StringLibrary.dll*:

 **Important**

If you are using the command line rather than Visual Studio to create the example, you should modify the call to the **ResourceManager** class constructor to `ResourceManager rm = new ResourceManager("Strings", typeof(Example).Assembly);`.

C#

```
using System;
using System.Globalization;
using System.Reflection;
using System.Resources;
using System.Threading;

[assembly:NeutralResourcesLanguageAttribute("en")]

public class StringLibrary
{
    public string GetGreeting()
    {
        ResourceManager rm = new ResourceManager("Strings",
Assembly.GetAssembly(typeof(StringLibrary)));
        string greeting = rm.GetString("Greeting");
        return greeting;
    }
}
```

The command for the C# compiler is:

Console

```
csc -t:library -resource:Strings.resources -delaysign+ -
keyfile:publickey.snk StringLibrary.cs
```

The corresponding Visual Basic compiler command is:

Console

```
vbc -t:library -resource:Strings.resources -delaysign+ -  
keyfile:publickey.snk StringLibrary.vb
```

8. Create a subdirectory in the main application directory for each localized culture supported by the application. You should create an *en-US*, an *fr-FR*, and an *ru-RU* subdirectory. Visual Studio creates these subdirectories automatically as part of the compilation process. Because all satellite assemblies have the same file name, the subdirectories are used to store individual culture-specific satellite assemblies until they're signed with a public/private key pair.
9. Embed the individual culture-specific *.resources* files into delay signed satellite assemblies and save them to the appropriate directory. The command to do this for each *.resources* file is:

Console

```
al -target:lib -embed:Strings.culture.resources -culture:culture -  
out:culture\StringLibrary.resources.dll -delay+ -keyfile:publickey.snk
```

where *culture* is the name of a culture. In this example, the culture names are *en-US*, *fr-FR*, and *ru-RU*.

10. Re-sign *StringLibrary.dll* by using the [Strong Name tool \(*sn.exe*\)](#) as follows:

Console

```
sn -R StringLibrary.dll RealKeyValuePair.snk
```

11. Re-sign the individual satellite assemblies. To do this, use the [Strong Name tool \(*sn.exe*\)](#) as follows for each satellite assembly:

Console

```
sn -R StringLibrary.resources.dll RealKeyValuePair.snk
```

12. Register *StringLibrary.dll* and each of its satellite assemblies in the global assembly cache by using the following command:

Console

```
gacutil -i filename
```

where *filename* is the name of the file to register.

13. If you're using Visual Studio, create a new **Console Application** project named `Example`, add a reference to *StringLibrary.dll* and the following source code to it, and compile.

C#

```
using System;
using System.Globalization;
using System.Threading;

public class Example
{
    public static void Main()
    {
        string[] cultureNames = { "en-GB", "en-US", "fr-FR", "ru-RU" };
        Random rnd = new Random();
        string cultureName = cultureNames[rnd.Next(0,
cultureNames.Length)];
        Thread.CurrentThread.CurrentCulture =
CultureInfo.CreateSpecificCulture(cultureName);
        Console.WriteLine("The current UI culture is {0}",
                           Thread.CurrentThread.CurrentCulture.Name);
        StringLibrary strLib = new StringLibrary();
        string greeting = strLib.GetGreeting();
        Console.WriteLine(greeting);
    }
}
```

To compile from the command line, use the following command for the C# compiler:

Console

```
csc Example.cs -r:StringLibrary.dll
```

The command line for the Visual Basic compiler is:

Console

```
vbc Example.vb -r:StringLibrary.dll
```

14. Run *Example.exe*.

See also

- [Package and deploy resources](#)
- [Delay signing an assembly](#)
- [*al.exe* \(Assembly Linker\)](#)
- [*sn.exe* \(Strong Name tool\)](#)
- [*gacutil.exe* \(Global Assembly Cache tool\)](#)
- [Resources in .NET](#)

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback [here](#).

 [Open a documentation issue](#)

 [Provide product feedback](#)

Package and deploy resources in .NET apps

Article • 03/18/2023

Applications rely on the .NET Framework Resource Manager, represented by the [ResourceManager](#) class, to retrieve localized resources. The Resource Manager assumes that a hub and spoke model is used to package and deploy resources. The hub is the main assembly that contains the nonlocalizable executable code and the resources for a single culture, called the neutral or default culture. The default culture is the fallback culture for the application; it is the culture whose resources are used if localized resources cannot be found. Each spoke connects to a satellite assembly that contains the resources for a single culture, but does not contain any code.

There are several advantages to this model:

- You can incrementally add resources for new cultures after you have deployed an application. Because subsequent development of culture-specific resources can require a significant amount of time, this allows you to release your main application first, and deliver culture-specific resources at a later date.
- You can update and change an application's satellite assemblies without recompiling the application.
- An application needs to load only those satellite assemblies that contain the resources needed for a particular culture. This can significantly reduce the use of system resources.

However, there are also disadvantages to this model:

- You must manage multiple sets of resources.
- The initial cost of testing an application increases, because you must test several configurations. Note that in the long term it will be easier and less expensive to test one core application with several satellites, than to test and maintain several parallel international versions.

Resource naming conventions

When you package your application's resources, you must name them using the resource naming conventions that the common language runtime expects. The runtime identifies a resource by its culture name. Each culture is given a unique name, which is usually a combination of a two-letter, lowercase culture name associated with a language and, if required, a two-letter, uppercase subculture name associated with a

country or region. The subculture name follows the culture name, separated by a dash (-). Examples include ja-JP for Japanese as spoken in Japan, en-US for English as spoken in the United States, de-DE for German as spoken in Germany, or de-AT for German as spoken in Austria. See the [Language tag](#) column in the [list of language/region names supported by Windows](#). Culture names follow the standard defined by [BCP 47](#).

Note

There are some exceptions for the two-letter culture names, such as `zh-Hans` for Chinese (Simplified).

For more information, see [Create resource files](#) and [Create satellite assemblies](#).

The resource fallback process

The hub and spoke model for packaging and deploying resources uses a fallback process to locate appropriate resources. If an application requests a localized resource that's unavailable, the common language runtime searches the hierarchy of cultures for an appropriate fallback resource that most closely matches the user's application's request, and throws an exception only as a last resort. At each level of the hierarchy, if an appropriate resource is found, the runtime uses it. If the resource is not found, the search continues at the next level.

To improve lookup performance, apply the [NeutralResourcesLanguageAttribute](#) attribute to your main assembly, and pass it the name of the neutral language that will work with your main assembly.

.NET Framework resource fallback process

The .NET Framework resource fallback process involves the following steps:

Tip

You may be able to use the `<relativeBindForResources>` configuration element to optimize the resource fallback process and the process by which the runtime probes for resource assemblies. For more information, see [Optimizing the resource fallback process](#).

1. The runtime first checks the [global assembly cache](#) for an assembly that matches the requested culture for your application.

The global assembly cache can store resource assemblies that are shared by many applications. This frees you from having to include specific sets of resources in the directory structure of every application you create. If the runtime finds a reference to the assembly, it searches the assembly for the requested resource. If it finds the entry in the assembly, it uses the requested resource. If it doesn't find the entry, it continues the search.

2. The runtime next checks the directory of the currently executing assembly for a subdirectory that matches the requested culture. If it finds the subdirectory, it searches that subdirectory for a valid satellite assembly for the requested culture. The runtime then searches the satellite assembly for the requested resource. If it finds the resource in the assembly, it uses it. If it doesn't find the resource, it continues the search.
3. The runtime next queries the Windows Installer to determine whether the satellite assembly is to be installed on demand. If so, it handles the installation, loads the assembly, and searches it for the requested resource. If it finds the resource in the assembly, it uses it. If it doesn't find the resource, it continues the search.
4. The runtime raises the [AppDomain.AssemblyResolve](#) event to indicate that it is unable to find the satellite assembly. If you choose to handle the event, your event handler can return a reference to the satellite assembly whose resources will be used for the lookup. Otherwise, the event handler returns `null` and the search continues.
5. The runtime next searches the global assembly cache again, this time for the parent assembly of the requested culture. If the parent assembly exists in the global assembly cache, the runtime searches the assembly for the requested resource.

The parent culture is defined as the appropriate fallback culture. Consider parents as fallback candidates, because providing any resource is preferable to throwing an exception. This process also allows you to reuse resources. You should include a particular resource at the parent level only if the child culture doesn't need to localize the requested resource. For example, if you supply satellite assemblies for `en` (neutral English), `en-GB` (English as spoken in the United Kingdom), and `en-US` (English as spoken in the United States), the `en` satellite would contain the common terminology, and the `en-GB` and `en-US` satellites could provide overrides for only those terms that differ.

6. The runtime next checks the directory of the currently executing assembly to see if it contains a parent directory. If a parent directory exists, the runtime searches the

directory for a valid satellite assembly for the parent culture. If it finds the assembly, the runtime searches the assembly for the requested resource. If it finds the resource, it uses it. If it doesn't find the resource, it continues the search.

7. The runtime next queries the Windows Installer to determine whether the parent satellite assembly is to be installed on demand. If so, it handles the installation, loads the assembly, and searches it or the requested resource. If it finds the resource in the assembly, it uses it. If it doesn't find the resource, it continues the search.
8. The runtime raises the [AppDomain.AssemblyResolve](#) event to indicate that it is unable to find an appropriate fallback resource. If you choose to handle the event, your event handler can return a reference to the satellite assembly whose resources will be used for the lookup. Otherwise, the event handler returns `null` and the search continues.
9. The runtime next searches parent assemblies, as in the previous three steps, through many potential levels. Each culture has only one parent, which is defined by the [CultureInfo.Parent](#) property, but a parent might have its own parent. The search for parent cultures stops when a culture's [Parent](#) property returns [CultureInfo.InvariantCulture](#); for resource fallback, the invariant culture is not considered a parent culture or a culture that can have resources.
10. If the culture that was originally specified and all parents have been searched and the resource is still not found, the resource for the default (fallback) culture is used. Typically, the resources for the default culture are included in the main application assembly. However, you can specify a value of [Satellite](#) for the [Location](#) property of the [NeutralResourcesLanguageAttribute](#) attribute to indicate that the ultimate fallback location for resources is a satellite assembly, rather than the main assembly.

Note

The default resource is the only resource that can be compiled with the main assembly. Unless you specify a satellite assembly by using the [NeutralResourcesLanguageAttribute](#) attribute, it is the ultimate fallback (final parent). Therefore, we recommend that you always include a default set of resources in your main assembly. This helps prevent exceptions from being thrown. By including a default resource, file you provide a fallback for all resources and ensure that at least one resource is always present for the user, even if it is not culturally specific.

11. Finally, if the runtime doesn't find a resource for a default (fallback) culture, a [MissingManifestResourceException](#) or [MissingSatelliteAssemblyException](#) exception is thrown to indicate that the resource could not be found.

For example, suppose the application requests a resource localized for Spanish (Mexico) (the `es-MX` culture). The runtime first searches the global assembly cache for the assembly that matches `es-MX`, but doesn't find it. The runtime then searches the directory of the currently executing assembly for an `es-MX` directory. Failing that, the runtime searches the global assembly cache again for a parent assembly that reflects the appropriate fallback culture — in this case, `es` (Spanish). If the parent assembly is not found, the runtime searches all potential levels of parent assemblies for the `es-MX` culture until it finds a corresponding resource. If a resource isn't found, the runtime uses the resource for the default culture.

Optimize the .NET Framework resource fallback process

Under the following conditions, you can optimize the process by which the runtime searches for resources in satellite assemblies:

- Satellite assemblies are deployed in the same location as the code assembly. If the code assembly is installed in the [Global Assembly Cache](#), satellite assemblies are also installed in the global assembly cache. If the code assembly is installed in a directory, satellite assemblies are installed in culture-specific folders of that directory.
- Satellite assemblies are not installed on demand.
- Application code does not handle the [AppDomain.AssemblyResolve](#) event.

You optimize the probe for satellite assemblies by including the `<relativeBindForResources>` element and setting its `enabled` attribute to `true` in the application configuration file, as shown in the following example.

XML

```
<configuration>
  <runtime>
    <relativeBindForResources enabled="true" />
  </runtime>
</configuration>
```

The optimized probe for satellite assemblies is an opt-in feature. That is, the runtime follows the steps documented in [The resource fallback process](#) unless the

`<relativeBindForResources>` element is present in the application's configuration file and its `enabled` attribute is set to `true`. If this is the case, the process of probing for a satellite assembly is modified as follows:

- The runtime uses the location of the parent code assembly to probe for the satellite assembly. If the parent assembly is installed in the global assembly cache, the runtime probes in the cache but not in the application's directory. If the parent assembly is installed in an application directory, the runtime probes in the application directory but not in the global assembly cache.
- The runtime doesn't query the Windows Installer for on-demand installation of satellite assemblies.
- If the probe for a particular resource assembly fails, the runtime does not raise the `AppDomain.AssemblyResolve` event.

.NET Core resource fallback process

The .NET Core resource fallback process involves the following steps:

1. The runtime attempts to load a satellite assembly for the requested culture.
 - Checks the directory of the currently executing assembly for a subdirectory that matches the requested culture. If it finds the subdirectory, it searches that subdirectory for a valid satellite assembly for the requested culture and loads it.

(!) Note

On operating systems with case-sensitive file systems (that is, Linux and macOS), the culture name subdirectory search is case-sensitive. The subdirectory name must exactly match the case of the `CultureInfo.Name` (for example, `es` or `es-MX`).

(!) Note

If the programmer has derived a custom assembly load context from `AssemblyLoadContext`, the situation is complicated. If the executing assembly was loaded into the custom context, the runtime loads the satellite assembly into the custom context. The details are out of scope for this document. See `AssemblyLoadContext`.

- If a satellite assemble has not been found, the [AssemblyLoadContext](#) raises the [AssemblyLoadContext.Resolving](#) event to indicate that it is unable to find the satellite assembly. If you choose to handle the event, your event handler can load and return a reference to the satellite assembly.
- If a satellite assembly still has not been found, the AssemblyLoadContext causes the AppDomain to trigger an [AppDomain.AssemblyResolve](#) event to indicate that it is unable to find the satellite assembly. If you choose to handle the event, your event handler can load and return a reference to the satellite assembly.

2. If a satellite assembly is found, the runtime searches it for the requested resource. If it finds the resource in the assembly, it uses it. If it doesn't find the resource, it continues the search.

 **Note**

To find a resource within the satellite assembly, the runtime searches for the resource file requested by the [ResourceManager](#) for the current [CultureInfo.Name](#). Within the resource file it searches for the requested resource name. If either is not found, the resource is treated as not found.

3. The runtime next searches the parent culture assemblies through many potential levels, each time repeating steps 1 & 2.

The parent culture is defined as an appropriate fallback culture. Consider parents as fallback candidates, because providing any resource is preferable to throwing an exception. This process also allows you to reuse resources. You should include a particular resource at the parent level only if the child culture doesn't need to localize the requested resource. For example, if you supply satellite assemblies for `en` (neutral English), `en-GB` (English as spoken in the United Kingdom), and `en-US` (English as spoken in the United States), the `en` satellite contains the common terminology, and the `en-GB` and `en-US` satellites provides overrides for only those terms that differ.

Each culture has only one parent, which is defined by the [CultureInfo.Parent](#) property, but a parent might have its own parent. The search for parent cultures stops when a culture's [Parent](#) property returns [CultureInfo.InvariantCulture](#). For resource fallback, the invariant culture is not considered a parent culture or a culture that can have resources.

4. If the culture that was originally specified and all parents have been searched and the resource is still not found, the resource for the default (fallback) culture is used. Typically, the resources for the default culture are included in the main application assembly. However, you can specify a value of [Satellite](#) for the [Location](#) property to indicate that the ultimate fallback location for resources is a satellite assembly rather than the main assembly.

 **Note**

The default resource is the only resource that can be compiled with the main assembly. Unless you specify a satellite assembly by using the [NeutralResourcesLanguageAttribute](#) attribute, it is the ultimate fallback (final parent). Therefore, we recommend that you always include a default set of resources in your main assembly. This helps prevent exceptions from being thrown. By including a default resource file, you provide a fallback for all resources and ensure that at least one resource is always present for the user, even if it is not culturally specific.

5. Finally, if the runtime doesn't find a resource file for a default (fallback) culture, a [MissingManifestResourceException](#) or [MissingSatelliteAssemblyException](#) exception is thrown to indicate that the resource could not be found. If the resource file is found but the requested resource is not present the request returns `null`.

Ultimate fallback to satellite assembly

You can optionally remove resources from the main assembly and specify that the runtime should load the ultimate fallback resources from a satellite assembly that corresponds to a specific culture. To control the fallback process, you use the [NeutralResourcesLanguageAttribute\(String, UltimateResourceFallbackLocation\)](#) constructor and supply a value for the [UltimateResourceFallbackLocation](#) parameter that specifies whether Resource Manager should extract the fallback resources from the main assembly or from a satellite assembly.

The following .NET Framework example uses the [NeutralResourcesLanguageAttribute](#) attribute to store an application's fallback resources in a satellite assembly for the French (`fr`) language. The example has two text-based resource files that define a single string resource named `Greeting`. The first, `resources.fr.txt`, contains a French language resource.

text

```
Greeting=Bonjour!
```

The second, resources.ru.txt, contains a Russian language resource.

```
text
```

```
Greeting=Добрый день
```

These two files are compiled to .resources files by running [Resource File Generator \(resgen.exe\)](#) from the command line. For the French language resource, the command is:

```
Console
```

```
resgen.exe resources.fr.txt
```

For the Russian language resource, the command is:

```
Console
```

```
resgen.exe resources.ru.txt
```

The .resources files are embedded into dynamic link libraries by running [Assembly Linker \(al.exe\)](#) from the command line for the French language resource as follows:

```
Console
```

```
al /t:lib /embed:resources.fr.resources /culture:fr  
/out:fr\Example1.resources.dll
```

and for the Russian language resource as follows:

```
Console
```

```
al /t:lib /embed:resources.ru.resources /culture:ru  
/out:ru\Example1.resources.dll
```

The application source code resides in a file named Example1.cs or Example1.vb. It includes the [NeutralResourcesLanguageAttribute](#) attribute to indicate that the default application resource is in the fr subdirectory. It instantiates the Resource Manager, retrieves the value of the `Greeting` resource, and displays it to the console.

```
C#
```

```
using System;
using System.Reflection;
using System.Resources;

[assembly:NeutralResourcesLanguage("fr",
UltimateResourceFallbackLocation.Satellite)]

public class Example
{
    public static void Main()
    {
        ResourceManager rm = new ResourceManager("resources",
                                                typeof(Example).Assembly);
        string greeting = rm.GetString("Greeting");
        Console.WriteLine(greeting);
    }
}
```

You can then compile C# source code from the command line as follows:

```
Console
csc Example1.cs
```

The command for the Visual Basic compiler is very similar:

```
Console
vbc Example1.vb
```

Because there are no resources embedded in the main assembly, you do not have to compile by using the `/resource` switch.

When you run the example from a system whose language is anything other than Russian, it displays the following output:

```
Output
Bon jour!
```

Suggested packaging alternative

Time or budget constraints might prevent you from creating a set of resources for every subculture that your application supports. Instead, you can create a single satellite assembly for a parent culture that all related subcultures can use. For example, you can

provide a single English satellite assembly (en) that is retrieved by users who request region-specific English resources, and a single German satellite assembly (de) for users who request region-specific German resources. For example, requests for German as spoken in Germany (de-DE), Austria (de-AT), and Switzerland (de-CH) would fall back to the German satellite assembly (de). The default resources are the final fallback and therefore should be the resources that will be requested by the majority of your application's users, so choose these resources carefully. This approach deploys resources that are less culturally specific, but can significantly reduce your application's localization costs.

See also

- [Resources in .NET apps](#)
- [Global Assembly Cache](#)
- [Create resource files](#)
- [Create satellite assemblies](#)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

Retrieve resources in .NET apps

Article • 03/18/2023

When you work with localized resources in .NET apps, you should ideally package the resources for the default or neutral culture with the main assembly and create a separate satellite assembly for each language or culture that your app supports. You can then use the [ResourceManager](#) class as described in the next section to access named resources. If you choose not to embed your resources in the main assembly and satellite assemblies, you can also access binary *.resources* files directly, as discussed in the section [Retrieve resources from .resources files](#) later in this article.

Retrieve resources from assemblies

The [ResourceManager](#) class provides access to resources at run time. You use the [ResourceManager.GetString](#) method to retrieve string resources and the [ResourceManager.GetObject](#) or [ResourceManager.GetStream](#) method to retrieve non-string resources. Each method has two overloads:

- An overload whose single parameter is a string that contains the name of the resource. The method attempts to retrieve that resource for the current culture. For more information, see the [GetString\(String\)](#), [GetObject\(String\)](#), and [GetStream\(String\)](#) methods.
- An overload that has two parameters: a string containing the name of the resource, and a [CultureInfo](#) object that represents the culture whose resource is to be retrieved. If a resource set for that culture cannot be found, the resource manager uses fallback rules to retrieve an appropriate resource. For more information, see the [GetString\(String, CultureInfo\)](#), [GetObject\(String, CultureInfo\)](#), and [GetStream\(String, CultureInfo\)](#) methods.

The resource manager uses the resource fallback process to control how the app retrieves culture-specific resources. For more information, see the "Resource Fallback Process" section in [Package and deploy resources](#). For information about instantiating a [ResourceManager](#) object, see the "Instantiating a ResourceManager Object" section in the [ResourceManager](#) class topic.

Retrieve string data example

The following example calls the [GetString\(String\)](#) method to retrieve the string resources of the current UI culture. It includes a neutral string resource for the English (United

States) culture and localized resources for the French (France) and Russian (Russia) cultures. The following English (United States) resource is in a file named Strings.txt:

```
text
```

```
TimeHeader=The current time is
```

The French (France) resource is in a file named Strings.fr-FR.txt:

```
text
```

```
TimeHeader=L'heure actuelle est
```

The Russian (Russia) resource is in a file named Strings.ru-RU.txt:

```
text
```

```
TimeHeader=Текущее время –
```

The source code for this example, which is in a file named GetString.cs for the C# version of the code and GetString.vb for the Visual Basic version, defines a string array that contains the name of four cultures: the three cultures for which resources are available and the Spanish (Spain) culture. A loop that executes five times randomly selects one of these cultures and assigns it to the [Thread.CurrentCulture](#) and [CultureInfo.CurrentUICulture](#) properties. It then calls the [GetString\(String\)](#) method to retrieve the localized string, which it displays along with the time of day.

```
C#
```

```
using System;
using System.Globalization;
using System.Resources;
using System.Threading;

[assembly: NeutralResourcesLanguageAttribute("en-US")]

public class Example
{
    public static void Main()
    {
        string[] cultureNames = { "en-US", "fr-FR", "ru-RU", "es-ES" };
        Random rnd = new Random();
        ResourceManager rm = new ResourceManager("Strings",
                                                typeof(Example).Assembly);

        for (int ctr = 0; ctr <= cultureNames.Length; ctr++) {
            string cultureName = cultureNames[rnd.Next(0,
```

```

cultureNames.Length)];
    CultureInfo culture =
CultureInfo.CreateSpecificCulture(cultureName);
    Thread.CurrentThread.CurrentCulture = culture;
    Thread.CurrentThread.CurrentUICulture = culture;

    Console.WriteLine("Current culture: {0}", culture.NativeName);
    string timeString = rm.GetString("TimeHeader");
    Console.WriteLine("{0} {1:T}\n", timeString, DateTime.Now);
}
}

// The example displays output like the following:
// Current culture: English (United States)
// The current time is 9:34:18 AM
//
// Current culture: Español (España, alfabetización internacional)
// The current time is 9:34:18
//
// Current culture: русский (Россия)
// Текущее время – 9:34:18
//
// Current culture: français (France)
// L'heure actuelle est 09:34:18
//
// Current culture: русский (Россия)
// Текущее время – 9:34:18

```

The following batch (.bat) file compiles the example and generates satellite assemblies in the appropriate directories. The commands are provided for the C# language and compiler. For Visual Basic, change `csc` to `vbc`, and change `GetString.cs` to `GetString.vb`.

```

Console

resgen strings.txt
csc GetString.cs -resource:string.resources

resgen strings.fr-FR.txt
md fr-FR
al -embed:string.resources -culture:fr-FR -out:fr-
FR\GetString.resources.dll

resgen strings.ru-RU.txt
md ru-RU
al -embed:string.resources -culture:ru-RU -out:ru-
RU\GetString.resources.dll

```

When the current UI culture is Spanish (Spain), note that the example displays English language resources, because Spanish language resources are unavailable, and English is

the example's default culture.

Retrieve object data examples

You can use the [GetObject](#) and [GetStream](#) methods to retrieve object data. This includes primitive data types, serializable objects, and objects that are stored in binary format (such as images).

The following example uses the [GetStream\(String\)](#) method to retrieve a bitmap that is used in an app's opening splash window. The following source code in a file named CreateResources.cs (for C#) or CreateResources.vb (for Visual Basic) generates a .resx file that contains the serialized image. In this case, the image is loaded from a file named SplashScreen.jpg; you can modify the file name to substitute your own image.

C#

```
using System;
using System.Drawing;
using System.Drawing.Imaging;
using System.IO;
using System.Resources;

public class Example
{
    public static void Main()
    {
        Bitmap bmp = new Bitmap(@".\SplashScreen.jpg");
        MemoryStream imageStream = new MemoryStream();
        bmp.Save(imageStream, ImageFormat.Jpeg);

        ResXResourceWriter writer = new
ResXResourceWriter("AppResources.resx");
        writer.AddResource("SplashScreen", imageStream);
        writer.Generate();
        writer.Close();
    }
}
```

The following code retrieves the resource and displays the image in a [PictureBox](#) control.

C#

```
using System;
using System.Drawing;
using System.IO;
using System.Resources;
using System.Windows.Forms;
```

```
public class Example
{
    public static void Main()
    {
        ResourceManager rm = new ResourceManager("AppResources",
typeof(Example).Assembly);
        Bitmap screen = (Bitmap)
Image.FromStream(rm.GetStream("SplashScreen"));

        Form frm = new Form();
        frm.Size = new Size(300, 300);

        PictureBox pic = new PictureBox();
        pic.Bounds = frm.RestoreBounds;
        pic.BorderStyle = BorderStyle.Fixed3D;
        pic.Image = screen;
        pic.SizeMode = PictureBoxSizeMode.StretchImage;

        frm.Controls.Add(pic);
        pic.Anchor = AnchorStyles.Top | AnchorStyles.Bottom |
            AnchorStyles.Left | AnchorStyles.Right;

        frm.ShowDialog();
    }
}
```

You can use the following batch file to build the C# example. For Visual Basic, change `csc` to `vbc`, and change the extension of the source code file from `.cs` to `.vb`.

```
Console

csc CreateResources.cs
CreateResources

resgen AppResources.resx

csc GetStream.cs -resource:AppResources.resources
```

The following example uses the [ResourceManager.GetObject\(String\)](#) method to deserialize a custom object. The example includes a source code file named `UIElements.cs` (`UIElements.vb` for Visual Basic) that defines the following structure named `PersonTable`. This structure is intended to be used by a general table display routine that displays the localized names of table columns. Note that the `PersonTable` structure is marked with the [SerializableAttribute](#) attribute.

C#

```
using System;

[Serializable] public struct PersonTable
{
    public readonly int nColumns;
    public readonly string column1;
    public readonly string column2;
    public readonly string column3;
    public readonly int width1;
    public readonly int width2;
    public readonly int width3;

    public PersonTable(string column1, string column2, string column3,
                      int width1, int width2, int width3)
    {
        this.column1 = column1;
        this.column2 = column2;
        this.column3 = column3;
        this.width1 = width1;
        this.width2 = width2;
        this.width3 = width3;
        this.nColumns = typeof(PersonTable).GetFields().Length / 2;
    }
}
```

The following code from a file named CreateResources.cs (CreateResources.vb for Visual Basic) creates an XML resource file named UIResources.resx that stores a table title and a `PersonTable` object that contains information for an app that is localized for the English language.

C#

```
using System;
using System.Resources;

public class CreateResource
{
    public static void Main()
    {
        PersonTable table = new PersonTable("Name", "Employee Number",
                                             "Age", 30, 18, 5);
        ResXResourceWriter rr = new ResXResourceWriter(@".\UIResources.resx");
        rr.AddResource("TableName", "Employees of Acme Corporation");
        rr.AddResource("Employees", table);
        rr.Generate();
        rr.Close();
    }
}
```

The following code in a source code file named GetObject.cs (GetObject.vb) then retrieves the resources and displays them to the console.

C#

```
using System;
using System.Resources;

[assembly: NeutralResourcesLanguageAttribute("en")]

public class Example
{
    public static void Main()
    {
        string fmtString = String.Empty;
        ResourceManager rm = new ResourceManager("UIResources",
typeof(Example).Assembly);
        string title = rm.GetString("TableName");
        PersonTable tableInfo = (PersonTable) rm.GetObject("Employees");

        if (! String.IsNullOrEmpty(title)) {
            fmtString = "{0," + ((Console.WindowWidth + title.Length) /
2).ToString() + "}";
            Console.WriteLine(fmtString, title);
            Console.WriteLine();
        }

        for (int ctr = 1; ctr <= tableInfo.nColumns; ctr++) {
            string columnName = "column" + ctr.ToString();
            string widthName = "width" + ctr.ToString();
            string value =
tableInfo.GetType().GetField(columnName).GetValue(tableInfo).ToString();
            int width = (int)
tableInfo.GetType().GetField(widthName).GetValue(tableInfo);
            fmtString = "{0,-" + width.ToString() + "}";
            Console.Write(fmtString, value);
        }
        Console.WriteLine();
    }
}
```

You can build the necessary resource file and assemblies and run the app by executing the following batch file. You must use the `/r` option to supply Resgen.exe with a reference to UIElements.dll so that it can access information about the `PersonTable` structure. If you're using C#, replace the `vbc` compiler name with `csc`, and replace the `.vb` extension with `.cs`.

Console

```
vbc -t:library UIElements.vb  
vbc CreateResources.vb -r:UIElements.dll  
CreateResources  
  
resgen UIResources.resx -r:UIElements.dll  
vbc GetObject.vb -r:UIElements.dll -resource:UIResources.resources  
  
GetObject.exe
```

Version support for satellite assemblies

By default, when the [ResourceManager](#) object retrieves requested resources, it looks for satellite assemblies that have version numbers that match the version number of the main assembly. After you have deployed an app, you might want to update the main assembly or specific resource satellite assemblies. The .NET Framework provides support for versioning the main assembly and satellite assemblies.

The [SatelliteContractVersionAttribute](#) attribute provides versioning support for a main assembly. Specifying this attribute on an app's main assembly enables you to update and redeploy a main assembly without updating its satellite assemblies. After you update the main assembly, increment the main assembly's version number but leave the satellite contract version number unchanged. When the resource manager retrieves requested resources, it loads the satellite assembly version specified by this attribute.

Publisher policy assemblies provide support for versioning satellite assemblies. You can update and redeploy a satellite assembly without updating the main assembly. After you update a satellite assembly, increment its version number and ship it with a publisher policy assembly. In the publisher policy assembly, specify that your new satellite assembly is backward-compatible with its previous version. The resource manager will use the [SatelliteContractVersionAttribute](#) attribute to determine the version of the satellite assembly, but the assembly loader will bind to the satellite assembly version specified by the publisher policy. For more information about publisher policy assemblies, see [Create a publisher policy file](#).

To enable full assembly versioning support, we recommend that you deploy strong-named assemblies in the [global assembly cache](#) and deploy assemblies that don't have strong names in the application directory. If you want to deploy strong-named assemblies in the application directory, you will not be able to increment a satellite assembly's version number when you update the assembly. Instead, you must perform an in-place update where you replace the existing code with the updated code and maintain the same version number. For example, if you want to update version 1.0.0.0 of a satellite assembly with the fully specified assembly name "myApp.resources,

Version=1.0.0.0, Culture=de, PublicKeyToken=b03f5f11d50a3a", overwrite it with the updated myApp.resources.dll that has been compiled with the same, fully specified assembly name "myApp.resources, Version=1.0.0.0, Culture=de, PublicKeyToken=b03f5f11d50a3a". Note that using in-place updates on satellite assembly files makes it difficult for an app to accurately determine the version of a satellite assembly.

For more information about assembly versioning, see [Assembly versioning](#) and [How the Runtime locates assemblies](#).

Retrieve resources from .resources Files

If you choose not to deploy resources in satellite assemblies, you can still use a [ResourceManager](#) object to access resources from .resources files directly. To do this, you must deploy the .resources files correctly. Then you use the [ResourceManager.CreateFileBasedResourceManager](#) method to instantiate a [ResourceManager](#) object and specify the directory that contains the standalone .resources files.

Deploy .resources Files

When you embed .resources files in an application assembly and satellite assemblies, each satellite assembly has the same file name, but is placed in a subdirectory that reflects the satellite assembly's culture. In contrast, when you access resources from .resources files directly, you can place all the .resources files in a single directory, usually a subdirectory of the application directory. The name of the app's default .resources file consists of a root name only, with no indication of its culture (for example, strings.resources). The resources for each localized culture are stored in a file whose name consists of the root name followed by the culture (for example, strings.ja.resources or strings.de-DE.resources).

The following illustration shows where resource files should be located in the directory structure. It also gives the naming conventions for .resource files.



Use the resource manager

After you have created your resources and placed them in the appropriate directory, you create a [ResourceManager](#) object to use the resources by calling the [CreateFileBasedResourceManager\(String, String, Type\)](#) method. The first parameter specifies the root name of the app's default .resources file (this would be "strings" for the example in the previous section). The second parameter specifies the location of the resources ("Resources" for the previous example). The third parameter specifies the [ResourceSet](#) implementation to use. If the third parameter is `null`, the default runtime [ResourceSet](#) is used.

ⓘ Note

Do not deploy ASP.NET apps using standalone .resources files. This can cause locking issues and breaks XCOPY deployment. We recommend that you deploy ASP.NET resources in satellite assemblies. For more information, see [ASP.NET Web Page Resources Overview](#).

After you instantiate the [ResourceManager](#) object, you use the [GetString](#), [GetObject](#), and [GetStream](#) methods as discussed earlier to retrieve the resources. However, the retrieval of resources directly from .resources files differs from the retrieval of embedded resources from assemblies. When you retrieve resources from .resources files, the [GetString\(String\)](#), [GetObject\(String\)](#), and [GetStream\(String\)](#) methods always retrieve the default culture's resources regardless of the current culture. To retrieve the resources of either the app's current culture or a specific culture, you must call the [GetString\(String, CultureInfo\)](#), [GetObject\(String, CultureInfo\)](#), or [GetStream\(String, CultureInfo\)](#) method and specify the culture whose resources are to be retrieved. To retrieve the resources of the current culture, specify the value of the [CultureInfo.CurrentCulture](#) property as the `culture` argument. If the resource manager cannot retrieve the resources of `culture`, it uses the standard resource fallback rules to retrieve the appropriate resources.

An example

The following example illustrates how the resource manager retrieves resources directly from .resources files. The example consists of three text-based resource files for the English (United States), French (France), and Russian (Russia) cultures. English (United States) is the example's default culture. Its resources are stored in the following file named *Strings.txt*:

```
text
```

```
Greeting=Hello  
Prompt=What is your name?
```

Resources for the French (France) culture are stored in the following file, which is named Strings.fr-FR.txt:

```
text  
  
Greeting=Bon jour  
Prompt=Comment vous appelez-vous?
```

Resources for the Russian (Russia) culture are stored in the following file, which is named Strings.ru-RU.txt:

```
text  
  
Greeting=Здравствуйте  
Prompt=Как вас зовут?
```

The following is the source code for the example. The example instantiates [CultureInfo](#) objects for the English (United States), English (Canada), French (France), and Russian (Russia) cultures, and makes each the current culture. The [ResourceManager.GetString\(String, CultureInfo\)](#) method then supplies the value of the [CultureInfo.CurrentCulture](#) property as the `culture` argument to retrieve the appropriate culture-specific resources.

```
C#  
  
using System;  
using System.Globalization;  
using System.Resources;  
using System.Threading;  
  
[assembly: NeutralResourcesLanguage("en-US")]  
  
public class Example  
{  
    public static void Main()  
    {  
        string[] cultureNames = { "en-US", "en-CA", "ru-RU", "fr-FR" };  
        ResourceManager rm =  
        ResourceManager.CreateFileBasedResourceManager("Strings", "Resources",  
        null);  
  
        foreach (var cultureName in cultureNames) {  
            Thread.CurrentThread.CurrentCulture =  
            CultureInfo.CreateSpecificCulture(cultureName);  
        }  
    }  
}
```

```

        string greeting = rm.GetString("Greeting",
CultureInfo.CurrentCulture);
        Console.WriteLine("\n{0}!", greeting);
        Console.Write(rm.GetString("Prompt", CultureInfo.CurrentCulture));
        string name = Console.ReadLine();
        if (! String.IsNullOrEmpty(name))
            Console.WriteLine("{0}, {1}!", greeting, name);
    }
    Console.WriteLine();
}
}

// The example displays output like the following:
//      Hello!
//      What is your name? Dakota
//      Hello, Dakota!

//
//      Hello!
//      What is your name? Koani
//      Hello, Koani!

//
//      Здравствуйте!
//      Как вас зовут?Samuel
//      Здравствуйте, Samuel!

//
//      Bon jour!
//      Comment vous appelez-vous?Yiska
//      Bon jour, Yiska!

```

You can compile the C# version of the example by running the following batch file. If you're using Visual Basic, replace `csc` with `vbc`, and replace the `.cs` extension with `.vb`.

Console

```

md Resources
resgen Strings.txt Resources\Strings.resources
resgen Strings.fr-FR.txt Resources\Strings.fr-FR.resources
resgen Strings.ru-RU.txt Resources\Strings.ru-RU.resources

csc Example.cs

```

See also

- [ResourceManager](#)
- [Resources in .NET apps](#)
- [Package and deploy resources](#)
- [How the Runtime locates assemblies](#)

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

System.Resources.MissingManifestResourceException class

Article • 01/08/2024

This article provides supplementary remarks to the reference documentation for this API.

A [MissingManifestResourceException](#) exception is thrown for different reasons in .NET versus UWP apps.

.NET apps

In .NET apps, [MissingManifestResourceException](#) is thrown when the attempt to retrieve a resource fails because the resource set for the neutral culture could not be loaded from a particular assembly. Although the exception is thrown when you try to retrieve a particular resource, it is caused by the failure to load the resource set rather than the failure to find the resource.

Note

For additional information, see the "Handling a MissingManifestResourceException Exception" section in the [ResourceManager](#) class topic.

The main causes of the exception include the following:

- The resource set is not identified by its fully qualified name. For example, if the `baseName` parameter in the call to the [ResourceManager.ResourceManager\(String, Assembly\)](#) method specifies the root name of the resource set without a namespace, but the resource set is assigned a namespace when it is stored in its assembly, the call to the [ResourceManager.GetString](#) method throws this exception.

If you've embedded the .resources file that contains the default culture's resources in your executable and your app is throwing a [MissingManifestResourceException](#), you can use a reflection tool such as the [IL Disassembler \(Ildasm.exe\)](#) to determine the fully qualified name of the resource. In ILDasm, double click the executable's **MANIFEST** label to open the **MANIFEST** window. Resources appear as `.mresource` items and are listed after external assembly references and custom assembly-level attributes. You can also compile the following simple utility, which lists the fully

qualified names of embedded resources in the assembly whose name is passed to it as a command-line parameter.

```
C#  
  
using System;  
using System.IO;  
using System.Reflection;  
using System.Resources;  
  
public class Example  
{  
    public static void Main()  
    {  
        if (Environment.GetCommandLineArgs().Length == 1) {  
            Console.WriteLine("No filename.");  
            return;  
        }  
  
        string filename = Environment.GetCommandLineArgs()[1].Trim();  
        // Check whether the file exists.  
        if (!File.Exists(filename)) {  
            Console.WriteLine("{0} does not exist.", filename);  
            return;  
        }  
  
        // Try to load the assembly.  
        Assembly assem = Assembly.LoadFrom(filename);  
        Console.WriteLine("File: {0}", filename);  
  
        // Enumerate the resource files.  
        string[] resNames = assem.GetManifestResourceNames();  
        if (resNames.Length == 0)  
            Console.WriteLine("No resources found.");  
  
        foreach (var resName in resNames)  
            Console.WriteLine("Resource: {0}",  
                resName.Replace(".resources", ""));  
  
        Console.WriteLine();  
    }  
}
```

- You identify the resource set by its resource file name (along with its optional namespace) and its file extension rather than by its namespace and root file name alone. For example, this exception is thrown if the neutral culture's resource set is named `GlobalResources` and you supply a value of `GlobalResources.resources` (instead of `GlobalResources`) to the `baseName` parameter of the `ResourceManager.ResourceManager(String, Assembly)` constructor.

- The culture-specific resource set that is identified in a method call cannot be found, and the fallback resource set cannot be loaded. For example, if you create satellite assemblies for the English (United States) and Russia (Russian) cultures but you fail to provide a resource set for the neutral culture, this exception is thrown if your app's current culture is English (United Kingdom).

[MissingManifestResourceException](#) uses the HRESULT `COR_E_MISSINGMANIFESTRESOURCE`, which has the value 0x80131532.

[MissingManifestResourceException](#) uses the default `Equals` implementation, which supports reference equality.

For a list of initial property values for an instance of [MissingManifestResourceException](#), see the [MissingManifestResourceException](#) constructors.

 **Note**

We recommend that you include a neutral set of resources in your main assembly, so your app won't fail if a satellite assembly is unavailable.

Universal Windows Platform (UWP) apps

UWP apps deploy resources for multiple cultures, including the neutral culture, in a single package resource index (.pri) file. As a result, in a UWP app, if resources for the preferred culture cannot be found, the [MissingManifestResourceException](#) is thrown under either of the following conditions:

- The app does not include a .pri file, or the .pri file could not be opened.
- The app's .pri file does not contain a resource set for the given root name.

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



.NET feedback

.NET is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

System.Resources.NeutralResourcesLanguageAttribute class

Article • 01/08/2024

This article provides supplementary remarks to the reference documentation for this API.

In desktop apps, the [NeutralResourcesLanguageAttribute](#) attribute informs the resource manager of an app's default culture and the location of its resources. By default, resources are embedded in the main app assembly, and you can use the attribute as follows. This statement specifies that the English (United States) is the app's default culture.

C#

```
[assembly: NeutralResourcesLanguage("en-US")]
```

You can also use the [NeutralResourcesLanguageAttribute](#) attribute to indicate where [ResourceManager](#) can find the resources of the default culture by providing an [UltimateResourceFallbackLocation](#) enumeration value in the attribute statement. This is most commonly done to indicate that the resources reside in a satellite assembly. For example, the following statement specifies that English (United States) is the app's default or neutral culture and that its resources reside in a satellite assembly. The [ResourceManager](#) object will look for them in a subdirectory named en-US.

C#

```
[assembly: NeutralResourcesLanguage("en-US",
UltimateResourceFallbackLocation.Satellite)]
```

Tip

We recommend that you always use the [NeutralResourcesLanguageAttribute](#) attribute to define the default culture of your app.

The attribute performs two roles:

- If the default culture's resources are embedded in the app's main assembly and [ResourceManager](#) has to retrieve resources that belong to the same culture as the default culture, the [ResourceManager](#) automatically uses the resources located in

the main assembly instead of searching for a satellite assembly. This bypasses the usual assembly probe, improves lookup performance for the first resource you load, and can reduce your working set. See [Packaging and Deploying Resources](#) for the process [ResourceManager](#) uses to probe for resource files.

- If the default culture's resources are located in a satellite assembly rather than in the main app assembly, the [NeutralResourcesLanguageAttribute](#) attribute specifies the culture and the directory from which the runtime can load the resources.

Windows 8.x Store apps

In Windows 8.x Store apps whose resources are loaded and retrieved by using the [ResourceManager](#) class, the [NeutralResourcesLanguageAttribute](#) attribute defines the neutral culture whose resources are used in the event of a failed probe. It does not specify the location of the resources. By default, [ResourceManager](#) uses the app's package resource index (PRI) file to locate the resources of the default culture. The neutral culture that is defined by the [NeutralResourcesLanguageAttribute](#) attribute is added to the end of the UI language list to simulate this effect.

If you load and retrieve resources by using the Windows Runtime [Windows.ApplicationModel.Resources.ResourceLoader](#) class or the types in the [Windows.ApplicationModel.Resources.Core](#) namespace, the [NeutralResourcesLanguageAttribute](#) attribute is ignored.

Examples

The following example uses a simple "Hello World" app to illustrate the use of the [NeutralResourcesLanguageAttribute](#) attribute to define a default or fallback culture. It requires the creation of separate resource files for the English (en), English (United States) (en-US), and French (France) (fr-FR) cultures. The following shows the contents of a text file named ExampleResources.txt for the English culture.

```
# Resources for the default (en) culture.  
Greeting=Hello
```

To use the resource file in an app, you must use the [Resource File Generator \(Resgen.exe\)](#) to convert the file from its text (.txt) format to a binary (.resources) format as follows:

```
resgen ExampleResources.txt
```

When the app is compiled, the binary resource file will be embedded in the main app assembly.

The following shows the contents of a text file named ExampleResources.en-US.txt that provides resources for the English (United States) culture.

```
# Resources for the en-US culture.  
Greeting=Hi
```

The text file can be converted to a binary resources file by using the [Resource File Generator \(ResGen.exe\)](#) at the command line as follows:

```
resgen ExampleResources.en-US.txt ExampleResources.en-US.resources
```

The binary resource file should then be compiled into an assembly by using [Assembly Linker \(Al.exe\)](#) and placed in the en-US subdirectory of the app directory by issuing the following command:

```
al /t:lib /embed:ExampleResources.en-US.resources /culture:en-US /out:en-US\Example.resources.dll
```

The following shows the contents of a text file named ExampleResources.fr-FR.txt that provides resources for the French (France) culture.

```
# Resources for the fr-FR culture.  
Greeting=Bonjour
```

The text file can be converted to a binary resource file by using ResGen.exe at the command line as follows:

```
resgen ExampleResources.fr-FR.txt ExampleResources.fr-FR.resources
```

The binary resources file should then be compiled into an assembly by using Assembly Linker and placed in the fr-FR subdirectory of the app directory by issuing the following command:

```
al /t:lib /embed:ExampleResources.fr-FR.resources /culture:fr-FR /out:fr-FR\Example.resources.dll
```

The following example provides the executable code that sets the current culture, prompts for the user's name, and displays a localized string.

C#

```
using System;
using System.Globalization;
using System.Reflection;
using System.Resources;
using System.Threading;

[assembly: NeutralResourcesLanguageAttribute("en")]
public class Example
{
    public static void Main()
    {
        // Select the current culture randomly to test resource fallback.
        string[] cultures = { "de-DE", "en-us", "fr-FR" };
        Random rnd = new Random();
        int index = rnd.Next(0, cultures.Length);
        Thread.CurrentThread.CurrentCulture =
CultureInfo.CreateSpecificCulture(cultures[index]);
        Console.WriteLine("The current culture is {0}",
                           CultureInfo.CurrentCulture.Name);

        // Retrieve the resource.
        ResourceManager rm = new ResourceManager("ExampleResources",
                                                typeof(Example).Assembly);
        string greeting = rm.GetString("Greeting");

        Console.Write("Enter your name: ");
        string name = Console.ReadLine();
        Console.WriteLine("{0} {1}!", greeting, name);
    }
}
```

It can be compiled by using the following command in Visual Basic:

```
vbc Example.vb /resource:ExampleResources.resources
```

or by using the following command in C#:

```
csc Example.cs /resource:ExampleResources.resources
```

 **Collaborate with us on GitHub**

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

.NET is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

System.Resources.ResourceManager class

Article • 01/08/2024

This article provides supplementary remarks to the reference documentation for this API.

ⓘ Important

Calling methods from this class with untrusted data is a security risk. Call the methods from this class only with trusted data. For more information, see [Validate All Inputs](#).

The [ResourceManager](#) class retrieves resources from a binary .resources file that is embedded in an assembly or from a standalone .resources file. If an app has been localized and localized resources have been deployed in [satellite assemblies](#), it looks up culture-specific resources, provides resource fallback when a localized resource does not exist, and supports resource serialization.

Desktop apps

For desktop apps, the [ResourceManager](#) class retrieves resources from binary resource (.resources) files. Typically, a language compiler or the [Assembly Linker \(AL.exe\)](#) embeds these resource files in an assembly. You can also use a [ResourceManager](#) object to retrieve resources directly from a .resources file that is not embedded in an assembly, by calling the [CreateFileBasedResourceManager](#) method.

✖ Caution

Using standalone .resources files in an ASP.NET app will break XCOPY deployment, because the resources remain locked until they are explicitly released by the [ReleaseAllResources](#) method. If you want to deploy resources with your ASP.NET apps, you should compile your .resources files into satellite assemblies.

In a resource-based app, one .resources file contains the resources of the default culture whose resources are used if no culture-specific resources can be found. For example, if an app's default culture is English (en), the English language resources are used whenever localized resources cannot be found for a specific culture, such as English

(United States) (en-US) or French (France) (fr-FR). Typically, the resources for the default culture are embedded in the main app assembly, and resources for other localized cultures are embedded in satellite assemblies. Satellite assemblies contain only resources. They have the same root file name as the main assembly and an extension of .resources.dll. For apps whose assemblies are not registered in the global assembly cache, satellite assemblies are stored in an app subdirectory whose name corresponds to the assembly's culture.

Create resources

When you develop a resource-based app, you store resource information in text files (files that have a .txt or .restext extension) or XML files (files that have a .resx extension). You then compile the text or XML files with the [Resource File Generator \(Resgen.exe\)](#) to create a binary .resources file. You can then embed the resulting .resources file in an executable or library by using a compiler option such as `/resources` for the C# and Visual Basic compilers, or you can embed it in a satellite assembly by using the [Assembly Linker \(Al.exe\)](#). If you include a .resx file in your Visual Studio project, Visual Studio handles the compilation and embedding of default and localized resources automatically as part of the build process.

Ideally, you should create resources for every language your app supports, or at least for a meaningful subset of each language. The binary .resources file names follow the naming convention *basename.cultureName.resources*, where *basename* is the name of the app or the name of a class, depending on the level of detail you want. The [CultureInfo.Name](#) property is used to determine *cultureName*. A resource for the app's default culture should be named *basename.resources*.

For example, suppose that an assembly has several resources in a resource file that has the base name MyResources. These resource files should have names such as MyResources.ja-JP.resources for the Japan (Japanese) culture, MyResources.de.resources for the German culture, MyResources.zh-CHS.resources for the simplified Chinese culture, and MyResources.fr-BE.resources for the French (Belgium) culture. The default resource file should be named MyResources.resources. The culture-specific resource files are commonly packaged in satellite assemblies for each culture. The default resource file should be embedded in the app's main assembly.

Note that [Assembly Linker](#) allows resources to be marked as private, but you should always mark them as public so they can be accessed by other assemblies. (Because a satellite assembly contains no code, resources that are marked as private are unavailable to your app through any mechanism.)

For more information about creating, packaging, and deploying resources, see the articles [Creating Resource Files](#), [Creating Satellite Assemblies](#), and [Packaging and Deploying Resources](#).

Instantiate a ResourceManager object

You instantiate a [ResourceManager](#) object that retrieves resources from an embedded .resources file by calling one of its class constructor overloads. This tightly couples a [ResourceManager](#) object with a particular .resources file and with any associated localized .resources files in satellite assemblies.

The two most commonly called constructors are:

- [ResourceManager\(String, Assembly\)](#) looks up resources based on two pieces of information that you supply: the base name of the .resources file, and the assembly in which the default .resources file resides. The base name includes the namespace and root name of the .resources file, without its culture or extension. Note that .resources files that are compiled from the command line typically do not include a namespace name, whereas .resources files that are created in the Visual Studio environment do. For example, if a resource file is named MyCompany.StringResources.resources and the [ResourceManager](#) constructor is called from a static method named `Example.Main`, the following code instantiates a [ResourceManager](#) object that can retrieve resources from the .resources file:

C#

```
ResourceManager rm = new ResourceManager("MyCompany.StringResources",
                                         typeof(Example).Assembly);
```

- [ResourceManager\(Type\)](#) looks up resources in satellite assemblies based on information from a type object. The type's fully qualified name corresponds to the base name of the .resources file without its file name extension. In desktop apps that are created by using the Visual Studio Resource Designer, Visual Studio creates a wrapper class whose fully qualified name is the same as the root name of the .resources file. For example, if a resource file is named MyCompany.StringResources.resources and there is a wrapper class named `MyCompany.StringResources`, the following code instantiates a [ResourceManager](#) object that can retrieve resources from the .resources file:

C#

```
ResourceManager rm = new
```

```
ResourceManager(typeof(MyCompany.StringResources));
```

If the appropriate resources cannot be found, the constructor call creates a valid `ResourceManager` object. However, the attempt to retrieve a resource throws a `MissingManifestResourceException` exception. For information about dealing with the exception, see the [Handle MissingManifestResourceException and MissingSatelliteAssembly Exceptions](#) section later in this article.

The following example shows how to instantiate a `ResourceManager` object. It contains the source code for an executable named ShowTime.exe. It also includes the following text file named Strings.txt that contains a single string resource, `TimeHeader`:

```
TimeHeader=The current time is
```

You can use a batch file to generate the resource file and embed it into the executable. Here's the batch file to generate an executable by using the C# compiler:

```
resgen strings.txt
csc ShowTime.cs /resource:strings.resources
```

For the Visual Basic compiler, you can use the following batch file:

```
resgen strings.txt
vbc ShowTime.vb /resource:strings.resources
```

C#

```
using System;
using System.Resources;

public class ShowTimeEx
{
    public static void Main()
    {
        ResourceManager rm = new ResourceManager("Strings",
                                                typeof(Example).Assembly);
        string timeString = rm.GetString("TimeHeader");
        Console.WriteLine("{0} {1:T}", timeString, DateTime.Now);
    }
}
```

```
// The example displays output like the following:  
//           The current time is 2:03:14 PM
```

ResourceManager and culture-specific resources

A localized app requires resources to be deployed, as discussed in the article [Packaging and Deploying Resources](#). If the assemblies are properly configured, the resource manager determines which resources to retrieve based on the current thread's [Thread.CurrentCulture](#) property. (That property also returns the current thread's UI culture.) For example, if an app is compiled with default English language resources in the main assembly and with French and Russian language resources in two satellite assemblies, and the [Thread.CurrentCulture](#) property is set to fr-FR, the resource manager retrieves the French resources.

You can set the [CurrentCulture](#) property explicitly or implicitly. The way you set it determines how the [ResourceManager](#) object retrieves resources based on culture:

- If you explicitly set the [Thread.CurrentCulture](#) property to a specific culture, the resource manager always retrieves the resources for that culture, regardless of the user's browser or operating system language. Consider an app that is compiled with default English language resources and three satellite assemblies that contain resources for English (United States), French (France), and Russian (Russia). If the [CurrentCulture](#) property is set to fr-FR, the [ResourceManager](#) object always retrieves the French (France) resources, even if the user's operating system language is not French. Make sure that this is the desired behavior before you set the property explicitly.

In ASP.NET apps, you must set the [Thread.CurrentCulture](#) property explicitly, because it is unlikely that the setting on the server will match incoming client requests. An ASP.NET app can set the [Thread.CurrentCulture](#) property explicitly to the user's browser accept language.

Explicitly setting the [Thread.CurrentCulture](#) property defines the current UI culture for that thread. It does not affect the current UI culture of any other threads in an app.

- You can set the UI culture of all threads in an app domain by assigning a [CultureInfo](#) object that represents that culture to the static [CultureInfo.DefaultThreadCurrentCulture](#) property.
- If you do not explicitly set the current UI culture and you do not define a default culture for the current app domain, the [CultureInfo.CurrentCulture](#) property is

set implicitly by the Windows `GetUserDefaultUILanguage` function. This function is provided by the Multilingual User Interface (MUI), which enables the user to set the default language. If the UI language is not set by the user, it defaults to the system-installed language, which is the language of operating system resources.

The following simple "Hello world" example sets the current UI culture explicitly. It contains resources for three cultures: English (United States) or en-US, French (France) or fr-FR, and Russian (Russia) or ru-RU. The en-US resources are contained in a text file named Greetings.txt:

```
HelloString=Hello world!
```

The fr-FR resources are contained in a text file named Greetings.fr-FR.txt:

```
HelloString=Salut tout le monde!
```

The ru-RU resources are contained in a text file named Greetings.ru-RU.txt:

```
HelloString=Всем привет!
```

Here's the source code for the example (Example.vb for the Visual Basic version or Example.cs for the C# version):

```
C#
```

```
using System;
using System.Globalization;
using System.Resources;
using System.Threading;

public class Example
{
    public static void Main()
    {
        // Create array of supported cultures
        string[] cultures = { "en-CA", "en-US", "fr-FR", "ru-RU" };
        Random rnd = new Random();
        int cultureNdx = rnd.Next(0, cultures.Length);
        CultureInfo originalCulture = Thread.CurrentThread.CurrentCulture;
        ResourceManager rm = new ResourceManager("Greetings",
typeof(Example).Assembly);
```

```

try
{
    CultureInfo newCulture = new CultureInfo(cultures[cultureNdx]);
    Thread.CurrentThread.CurrentCulture = newCulture;
    Thread.CurrentThread.CurrentUICulture = newCulture;
    string greeting = String.Format("The current culture is
{0}.\n{1}",

Thread.CurrentThread.CurrentUICulture.Name,
                                         rm.GetString("HelloString"));
    Console.WriteLine(greeting);
}
catch (CultureNotFoundException e)
{
    Console.WriteLine("Unable to instantiate culture {0}",
e.InvalidCultureName);
}
finally
{
    Thread.CurrentThread.CurrentCulture = originalCulture;
    Thread.CurrentThread.CurrentUICulture = originalCulture;
}
}

// The example displays output like the following:
//      The current culture is ru-RU.
//      Всем привет!

```

To compile this example, create a batch (.bat) file that contains the following commands and run it from the command prompt. If you're using C#, specify `csc` instead of `vbc` and `Example.cs` instead of `Example.vb`.

```

resgen Greetings.txt
vbc Example.vb /resource:Greetings.resources

resgen Greetings.fr-FR.txt
Md fr-FR
al /embed:Greetings.fr-FR.resources /culture:fr-FR /out:fr-
FR\Example.resources.dll

resgen Greetings.ru-RU.txt
Md ru-RU
al /embed:Greetings.ru-RU.resources /culture:ru-RU /out:ru-
RU\Example.resources.dll

```

Retrieve resources

You call the [GetObject\(String\)](#) and [GetString\(String\)](#) methods to access a specific resource. You can also call the [GetStream\(String\)](#) method to retrieve non-string resources as a byte array. By default, in an app that has localized resources, these methods return the resource for the culture determined by the current UI culture of the thread that made the call. See the previous section, [ResourceManager and culture-specific resources](#), for more information about how the current UI culture of a thread is defined. If the resource manager cannot find the resource for the current thread's UI culture, it uses a fallback process to retrieve the specified resource. If the resource manager cannot find any localized resources, it uses the resources of the default culture. For more information about resource fallback rules, see the "Resource Fallback Process" section of the article [Packaging and Deploying Resources](#).

Note

If the .resources file specified in the [ResourceManager](#) class constructor cannot be found, the attempt to retrieve a resource throws a [MissingManifestResourceException](#) or [MissingSatelliteAssemblyException](#) exception. For information about dealing with the exception, see the [Handle MissingManifestResourceException and MissingSatelliteAssemblyException Exceptions](#) section later in this article.

The following example uses the [GetString](#) method to retrieve culture-specific resources. It consists of resources compiled from .txt files for the English (en), French (France) (fr-FR), and Russian (Russia) (ru-RU) cultures. The example changes the current culture and current UI culture to English (United States), French (France), Russian (Russia), and Swedish (Sweden). It then calls the [GetString](#) method to retrieve the localized string, which it displays along with the current day and month. Notice that the output displays the appropriate localized string except when the current UI culture is Swedish (Sweden). Because Swedish language resources are unavailable, the app instead uses the resources of the default culture, which is English.

The example requires the text-based resource files listed in following table. Each has a single string resource named `DateStart`.

 Expand table

| Culture | File name | Resource name | Resource value |
|---------|-----------------------|---------------|-----------------------|
| en-US | DateStrings.txt | DateStart | Today is |
| fr-FR | DateStrings.fr-FR.txt | DateStart | Aujourd'hui, c'est le |

| Culture | File name | Resource name | Resource value |
|---------|-----------------------|---------------|----------------|
| ru-RU | DateStrings.ru-RU.txt | DateStart | Сегодня |

Here's the source code for the example (ShowDate.vb for the Visual Basic version or ShowDate.cs for the C# version of the code).

C#

```

using System;
using System.Globalization;
using System.Resources;
using System.Threading;

[assembly: NeutralResourcesLanguage("en")]

public class ShowDateEx
{
    public static void Main()
    {
        string[] cultureNames = { "en-US", "fr-FR", "ru-RU", "sv-SE" };
        ResourceManager rm = new ResourceManager("DateStrings",
                                                typeof(Example).Assembly);

        foreach (var cultureName in cultureNames)
        {
            CultureInfo culture =
CultureInfo.CreateSpecificCulture(cultureName);
            Thread.CurrentThread.CurrentCulture = culture;
            Thread.CurrentThread.CurrentUICulture = culture;

            Console.WriteLine("Current UI Culture: {0}",
                              CultureInfo.CurrentCulture.Name);
            string dateString = rm.GetString("DateStart");
            Console.WriteLine("{0} {1:M}.\\n", dateString, DateTime.Now);
        }
    }
}

// The example displays output similar to the following:
//      Current UI Culture: en-US
//      Today is February 03.
//
//      Current UI Culture: fr-FR
//      Aujourd'hui, c'est le 3 février
//
//      Current UI Culture: ru-RU
//      Сегодня февраля 03.
//
//      Current UI Culture: sv-SE
//      Today is den 3 februari.

```

To compile this example, create a batch file that contains the following commands and run it from the command prompt. If you're using C#, specify `csc` instead of `vbc` and `showdate.cs` instead of `showdate.vb`.

```
resgen DateStrings.txt
vbc showdate.vb /resource:DateStrings.resources

md fr-FR
resgen DateStrings.fr-FR.txt
al /out:fr-FR>Showdate.resources.dll /culture:fr-FR /embed:DateStrings.fr-
FR.resources

md ru-RU
resgen DateStrings.ru-RU.txt
al /out:ru-RU>Showdate.resources.dll /culture:ru-RU /embed:DateStrings.ru-
RU.resources
```

There are two ways to retrieve the resources of a specific culture other than the current UI culture:

- You can call the [GetString\(String, CultureInfo\)](#), [GetObject\(String, CultureInfo\)](#), or [GetStream\(String, CultureInfo\)](#) method to retrieve a resource for a specific culture. If a localized resource cannot be found, the resource manager uses the resource fallback process to locate an appropriate resource.
- You can call the [GetResourceSet](#) method to obtain a [ResourceSet](#) object that represents the resources for a particular culture. In the method call, you can determine whether the resource manager probes for parent cultures if it is unable to find localized resources, or whether it simply falls back to the resources of the default culture. You can then use the [ResourceSet](#) methods to access the resources (localized for that culture) by name, or to enumerate the resources in the set.

Handle [MissingManifestResourceException](#) and [MissingSatelliteAssemblyException](#) exceptions

If you try to retrieve a specific resource, but the resource manager cannot find that resource and either no default culture has been defined or the resources of the default culture cannot be located, the resource manager throws a [MissingManifestResourceException](#) exception if it expects to find the resources in the main assembly or a [MissingSatelliteAssemblyException](#) if it expects to find the resources in a satellite assembly. Note that the exception is thrown when you call a resource retrieval method such as [GetString](#) or [GetObject](#), and not when you instantiate a [ResourceManager](#) object.

The exception is typically thrown under the following conditions:

- The appropriate resource file or satellite assembly does not exist. If the resource manager expects the app's default resources to be embedded in the main app assembly, they are absent. If the [NeutralResourcesLanguageAttribute](#) attribute indicates that the app's default resources reside in a satellite assembly, that assembly cannot be found. When you compile your app, make sure that resources are embedded in the main assembly or that the necessary satellite assembly is generated and is named appropriately. Its name should take the form *appName.resources.dll*, and it should be located in a directory named after the culture whose resources it contains.
- Your app doesn't have a default or neutral culture defined. Add the [NeutralResourcesLanguageAttribute](#) attribute to a source code file or to the project information file (AssemblyInfo.vb for a Visual Basic app or AssemblyInfo.cs for a C# app) file.
- The `baseName` parameter in the [ResourceManager\(String, Assembly\)](#) constructor does not specify the name of a .resources file. The name should include the resource file's fully qualified namespace but not its file name extension. Typically, resource files that are created in Visual Studio include namespace names, but resource files that are created and compiled at the command prompt do not. You can determine the names of embedded .resources files by compiling and running the following utility. This is a console app that accepts the name of a main assembly or satellite assembly as a command-line parameter. It displays the strings that should be provided as the `baseName` parameter so that the resource manager can correctly identify the resource.

C#

```
using System;
using System.IO;
using System.Reflection;
using System.Resources;

public class Example
{
    public static void Main()
    {
        if (Environment.GetCommandLineArgs().Length == 1) {
            Console.WriteLine("No filename.");
            return;
        }

        string filename = Environment.GetCommandLineArgs()[1].Trim();
        // Check whether the file exists.
```

```

if (! File.Exists(filename)) {
    Console.WriteLine("{0} does not exist.", filename);
    return;
}

// Try to load the assembly.
Assembly assem = Assembly.LoadFrom(filename);
Console.WriteLine("File: {0}", filename);

// Enumerate the resource files.
string[] resNames = assem.GetManifestResourceNames();
if (resNames.Length == 0)
    Console.WriteLine("No resources found.");

foreach (var resName in resNames)
    Console.WriteLine("Resource: {0}",
        resName.Replace(".resources", ""));
}

Console.WriteLine();
}
}

```

If you are changing the current culture of your application explicitly, you should also remember that the resource manager retrieves a resource set based on the value of the [CultureInfo.CurrentCulture](#) property, and not the [CultureInfo.CurrentCulture](#) property. Typically, if you change one value, you should also change the other.

Resource versioning

Because the main assembly that contains an app's default resources is separate from the app's satellite assemblies, you can release a new version of your main assembly without redeploying the satellite assemblies. You use the [SatelliteContractVersionAttribute](#) attribute to use existing satellite assemblies and instruct the resource manager not to redeploy them with a new version of your main assembly.

For more information about versioning support for satellite assemblies, see the article [Retrieving Resources](#).

<satelliteassemblies> configuration file node

Note

This section is specific to .NET Framework apps.

For executables that are deployed and run from a website (HREF .exe files), the [ResourceManager](#) object may probe for satellite assemblies over the web, which can hurt your app's performance. To eliminate the performance problem, you can limit this probing to the satellite assemblies that you have deployed with your app. To do this, you create a `<satelliteassemblies>` node in your app's configuration file to specify that you have deployed a specific set of cultures for your app, and that the [ResourceManager](#) object should not try to probe for any culture that is not listed in that node.

 **Note**

The preferred alternative to creating a `<satelliteassemblies>` node is to use the [ClickOnce Deployment Manifest](#) feature.

In your app's configuration file, create a section similar to the following:

XML

```
<?xml version ="1.0"?>
<configuration>
  <satelliteassemblies>
    <assembly name="MainAssemblyName, Version=versionNumber,
Culture=neutral, PublicKeyToken=null|yourPublicKeyToken">
      <culture>cultureName1</culture>
      <culture>cultureName2</culture>
      <culture>cultureName3</culture>
    </assembly>
  </satelliteassemblies>
</configuration>
```

Edit this configuration information as follows:

- Specify one or more `<assembly>` nodes for each main assembly that you deploy, where each node specifies a fully qualified assembly name. Specify the name of your main assembly in place of *MainAssemblyName*, and specify the `Version`, `PublicKeyToken`, and `Culture` attribute values that correspond to your main assembly.

For the `Version` attribute, specify the version number of your assembly. For example, the first release of your assembly might be version number 1.0.0.0.

For the `PublicKeyToken` attribute, specify the keyword `null` if you have not signed your assembly with a strong name, or specify your public key token if you have signed your assembly.

For the `Culture` attribute, specify the keyword `neutral` to designate the main assembly and cause the [ResourceManager](#) class to probe only for the cultures listed in the `<culture>` nodes.

For more information about fully qualified assembly names, see the article [Assembly Names](#). For more information about strong-named assemblies, see the article [Create and use strong-named assemblies](#).

- Specify one or more `<culture>` nodes with a specific culture name, such as "fr-FR", or a neutral culture name, such as "fr".

If resources are needed for any assembly not listed under the `<satelliteassemblies>` node, the [ResourceManager](#) class probes for cultures using standard probing rules.

Windows 8.x apps

Important

Although the [ResourceManager](#) class is supported in Windows 8.x apps, we do not recommend its use. Use this class only when you develop Portable Class Library projects that can be used with Windows 8.x apps. To retrieve resources from Windows 8.x apps, use the [Windows.ApplicationModel.Resources.ResourceLoader](#) class instead.

For Windows 8.x apps, the [ResourceManager](#) class retrieves resources from package resource index (PRI) files. A single PRI file (the application package PRI file) contains the resources for both the default culture and any localized cultures. You use the MakePRI utility to create a PRI file from one or more resource files that are in XML resource (.resw) format. For resources that are included in a Visual Studio project, Visual Studio handles the process of creating and packaging the PRI file automatically. You can then use the .NET [ResourceManager](#) class to access the app's or library's resources.

You can instantiate a [ResourceManager](#) object for a Windows 8.x app in the same way that you do for a desktop app.

You can then access the resources for a particular culture by passing the name of the resource to be retrieved to the [GetString\(String\)](#) method. By default, this method returns the resource for the culture determined by the current UI culture of the thread that made the call. You can also retrieve the resources for a specific culture by passing the name of the resource and a [CultureInfo](#) object that represents the culture whose resource is to be retrieved to the [GetString\(String, CultureInfo\)](#) method. If the resource

for the current UI culture or the specified culture cannot be found, the resource manager uses a UI language fallback list to locate a suitable resource.

Examples

The following example demonstrates how to use an explicit culture and the implicit current UI culture to obtain string resources from a main assembly and a satellite assembly. For more information, see the "Directory Locations for Satellite Assemblies Not Installed in the Global Assembly Cache" section of the [Creating Satellite Assemblies](#) topic.

To run this example:

1. In the app directory, create a file named rmc.txt that contains the following resource strings:

```
day=Friday  
year=2006  
holiday="Cinco de Mayo"
```

2. Use the [Resource File Generator](#) to generate the rmc.resources resource file from the rmc.txt input file as follows:

```
resgen rmc.txt
```

3. Create a subdirectory of the app directory and name it "es-MX". This is the culture name of the satellite assembly that you will create in the next three steps.
4. Create a file named rmc.es-MX.txt in the es-MX directory that contains the following resource strings:

```
day=Viernes  
year=2006  
holiday="Cinco de Mayo"
```

5. Use the [Resource File Generator](#) to generate the rmc.es-MX.resources resource file from the rmc.es-MX.txt input file as follows:

```
resgen rmc.es-MX.txt
```

6. Assume that the filename for this example is rmc.vb or rmc.cs. Copy the following source code into a file. Then compile it and embed the main assembly resource file, rmc.resources, in the executable assembly. If you are using the Visual Basic compiler, the syntax is:

```
vbc rmc.vb /resource:rmc.resources
```

The corresponding syntax for the C# compiler is:

```
csc /resource:rmc.resources rmc.cs
```

7. Use the [Assembly Linker](#) to create a satellite assembly. If the base name of the app is rmc, the satellite assembly name must be rmc.resources.dll. The satellite assembly should be created in the es-MX directory. If es-MX is the current directory, use this command:

```
al /embed:rmc.es-MX.resources /c:es-MX /out:rmc.resources.dll
```

8. Run rmc.exe to obtain and display the embedded resource strings.

C#

```
using System;
using System.Globalization;
using System.Resources;

class Example2
{
    public static void Main()
    {
        string day;
        string year;
        string holiday;
        string celebrate = "{0} will occur on {1} in {2}.\n";
        // Create a resource manager.
```

```

ResourceManager rm = new ResourceManager("rmc",
                                         typeof(Example).Assembly);

Console.WriteLine("Obtain resources using the current UI culture.");

    // Get the resource strings for the day, year, and holiday
    // using the current UI culture.
    day = rm.GetString("day");
    year = rm.GetString("year");
    holiday = rm.GetString("holiday");
    Console.WriteLine(celebrate, holiday, day, year);

    // Obtain the es-MX culture.
    CultureInfo ci = new CultureInfo("es-MX");

    Console.WriteLine("Obtain resources using the es-MX culture.");

    // Get the resource strings for the day, year, and holiday
    // using the specified culture.
    day = rm.GetString("day", ci);
    year = rm.GetString("year", ci);
    holiday = rm.GetString("holiday", ci);
    // -----
    // Alternatively, comment the preceding 3 code statements and
    // uncomment the following 4 code statements:
    // -----
    // Set the current UI culture to "es-MX" (Spanish-Mexico).
    //     Thread.CurrentThread.CurrentCulture = ci;

    // Get the resource strings for the day, year, and holiday
    // using the current UI culture. Use those strings to
    // display a message.
    //     day = rm.GetString("day");
    //     year = rm.GetString("year");
    //     holiday = rm.GetString("holiday");
    // -----

    // Regardless of the alternative that you choose, display a message
    // using the retrieved resource strings.
    Console.WriteLine(celebrate, holiday, day, year);
}

/*
This example displays the following output:

```

Obtain resources using the current UI culture.
 "5th of May" will occur on Friday in 2006.

Obtain resources using the es-MX culture.
 "Cinco de Mayo" will occur on Viernes in 2006.

*/

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

.NET is an open source project.
Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

System.Resources.ResourceReader class

Article • 01/08/2024

This article provides supplementary remarks to the reference documentation for this API.

ⓘ Important

Calling methods from this class with untrusted data is a security risk. Call the methods from this class only with trusted data. For more information, see [Validate All Inputs](#).

The [ResourceReader](#) class provides a standard implementation of the [IResourceReader](#) interface. A [ResourceReader](#) instance represents either a standalone .resources file or a .resources file that is embedded in an assembly. It is used to enumerate the resources in a .resources file and retrieve its name/value pairs. It differs from the [ResourceManager](#) class, which is used to retrieve specified named resources from a .resources file that is embedded in an assembly. The [ResourceManager](#) class is used to retrieve resources whose names are known in advance, whereas the [ResourceReader](#) class is useful for retrieving resources whose number or precise names are not known at compile time. For example, an application may use a resources file to store configuration information that is organized into sections and items in a section, where the number of sections or items in a section is not known in advance. Resources can then be named generically (such as `Section1`, `Section1Item1`, `Section1Item2`, and so on) and retrieved by using a [ResourceReader](#) object.

ⓘ Important

This type implements the [IDisposable](#) interface. When you have finished using the type, you should dispose of it either directly or indirectly. To dispose of the type directly, call its [Dispose](#) method in a `try/catch` block. To dispose of it indirectly, use a language construct such as `using` (in C#) or `Using` (in Visual Basic). For more information, see the "Using an Object that Implements [IDisposable](#)" section in the [IDisposable](#) interface documentation.

Instantiate a ResourceReader object

A .resources file is a binary file that has been compiled from either a text file or an XML .resx file by Resgen.exe (Resource File Generator). A [ResourceReader](#) object can represent either a standalone .resources file or a .resources file that has been embedded in an assembly.

To instantiate a [ResourceReader](#) object that reads from a standalone .resources file, use the [ResourceReader](#) class constructor with either an input stream or a string that contains the .resources file name. The following example illustrates both approaches. The first instantiates a [ResourceReader](#) object that represents a .resources file named `Resources1.resources` by using its file name. The second instantiates a [ResourceReader](#) object that represents a .resources file named `Resources2.resources` by using a stream created from the file.

C#

```
// Instantiate a standalone .resources file from its filename.  
var rr1 = new System.Resources.ResourceReader("Resources1.resources");  
  
// Instantiate a standalone .resources file from a stream.  
var fs = new System.IO.FileStream(@".\Resources2.resources",  
                           System.IO FileMode.Open);  
var rr2 = new System.Resources.ResourceReader(fs);
```

To create a [ResourceReader](#) object that represents an embedded .resources file, instantiate an [Assembly](#) object from the assembly in which the .resources file is embedded. Its [Assembly.GetManifestResourceStream](#) method returns a [Stream](#) object that can be passed to the [ResourceReader\(Stream\)](#) constructor. The following example instantiates a [ResourceReader](#) object that represents an embedded .resources file.

C#

```
System.Reflection.Assembly assem =  
    System.Reflection.Assembly.LoadFrom(@".\MyLibrary.dll");  
System.IO.Stream fs =  
  
assem.GetManifestResourceStream("MyCompany.LibraryResources.resources");  
var rr = new System.Resources.ResourceReader(fs);
```

Enumerate a ResourceReader object's resources

To enumerate the resources in a .resources file, you call the [GetEnumerator](#) method, which returns an [System.Collections.IDictionaryEnumerator](#) object. You call the [IDictionaryEnumerator.MoveNext](#) method to move from one resource to the next. The

method returns `false` when all the resources in the .resources file have been enumerated.

ⓘ Note

Although the `ResourceReader` class implements the `IEnumerable` interface and the `IEnumerable.GetEnumerator` method, the `ResourceReader.GetEnumerator` method does not provide the `IEnumerable.GetEnumerator` implementation. Instead, the `ResourceReader.GetEnumerator` method returns an `IDictionaryEnumerator` interface object that provides access to each resource's name/value pair.

You can retrieve the individual resources in the collection in two ways:

- You can iterate each resource in the `System.Collections.IDictionaryEnumerator` collection and use `System.Collections.IDictionaryEnumerator` properties to retrieve the resource name and value. We recommend this technique when all the resources are of the same type, or you know the data type of each resource.
- You can retrieve the name of each resource when you iterate the `System.Collections.IDictionaryEnumerator` collection and call the `GetResourceData` method to retrieve the resource's data. We recommend this approach when you do not know the data type of each resource or if the previous approach throws exceptions.

Retrieve resources by using `IDictionaryEnumerator` properties

The first method of enumerating the resources in a .resources file involves directly retrieving each resource's name/value pair. After you call the `IDictionaryEnumerator.MoveNext` method to move to each resource in the collection, you can retrieve the resource name from the `IDictionaryEnumerator.Key` property and the resource data from the `IDictionaryEnumerator.Value` property.

The following example shows how to retrieve the name and value of each resource in a .resources file by using the `IDictionaryEnumerator.Key` and `IDictionaryEnumerator.Value` properties. To run the example, create the following text file named ApplicationResources.txt to define string resources.

```
Title="Contact Information"  
Label1="First Name:"
```

```
Label2="Middle Name:"  
Label3="Last Name:"  
Label4="SSN:"  
Label5="Street Address:"  
Label6="City:"  
Label7="State:"  
Label8="Zip Code:"  
Label9="Home Phone:"  
Label10="Business Phone:"  
Label11="Mobile Phone:"  
Label12="Other Phone:"  
Label13="Fax:"  
Label14="Email Address:"  
Label15="Alternate Email Address:"
```

You can then convert the text resource file to a binary file named ApplicationResources.resources by using the following command:

```
resgen ApplicationResources.txt
```

The following example then uses the [ResourceReader](#) class to enumerate each resource in the standalone binary .resources file and to display its key name and corresponding value.

C#

```
using System;  
using System.Collections;  
using System.Resources;  
  
public class Example1  
{  
    public static void Run()  
    {  
        Console.WriteLine("Resources in ApplicationResources.resources:");  
        ResourceReader res = new  
ResourceReader(@"..\ApplicationResources.resources");  
        IDictionaryEnumerator dict = res.Getenumerator();  
        while (dict.MoveNext())  
            Console.WriteLine("    {0}: '{1}' (Type {2})",  
                            dict.Key, dict.Value, dict.Value.GetType().Name);  
        res.Close();  
    }  
}  
// The example displays the following output:  
//     Resources in ApplicationResources.resources:  
//         Label3: '"Last Name"' (Type String)  
//         Label2: '"Middle Name"' (Type String)  
//         Label1: '"First Name"' (Type String)  
//         Label7: '"State"' (Type String)  
//         Label6: '"City"' (Type String)  
//         Label5: '"Street Address"' (Type String)
```

```
//      Label14: '"SSN:"' (Type String)
//      Label19: '"Home Phone:"' (Type String)
//      Label18: '"Zip Code:"' (Type String)
//      Title: '"Contact Information"' (Type String)
//      Label12: '"Other Phone:"' (Type String)
//      Label13: '"Fax:"' (Type String)
//      Label10: '"Business Phone:"' (Type String)
//      Label11: '"Mobile Phone:"' (Type String)
//      Label14: '"Email Address:"' (Type String)
//      Label15: '"Alternate Email Address:"' (Type String)
```

The attempt to retrieve resource data from the [IDictionaryEnumerator.Value](#) property can throw the following exceptions:

- A [FormatException](#) if the data is not in the expected format.
- A [FileNotFoundException](#) if the assembly that contains the type to which the data belongs cannot be found.
- A [TypeLoadException](#) if the type to which the data belongs cannot be found.

Typically, these exceptions are thrown if the .resources file has been modified manually, if the assembly in which a type is defined has either not been included with an application or has been inadvertently deleted, or if the assembly is an older version that predates a type. If one of these exceptions is thrown, you can retrieve resources by enumerating each resource and calling the [GetResourceData](#) method, as the following section shows. This approach provides you with some information about the data type that the [IDictionaryEnumerator.Value](#) property attempted to return.

Retrieve resources by name with GetResourceData

The second approach to enumerating resources in a .resources file also involves navigating through the resources in the file by calling the [IDictionaryEnumerator.MoveNext](#) method. For each resource, you retrieve the resource's name from the [IDictionaryEnumerator.Key](#) property, which is then passed to the [GetResourceData\(String, String, Byte\[\]\)](#) method to retrieve the resource's data. This is returned as a byte array in the [resourceData](#) argument.

This approach is more awkward than retrieving the resource name and value from the [IDictionaryEnumerator.Key](#) and [IDictionaryEnumerator.Value](#) properties, because it returns the actual bytes that form the resource value. However, if the attempt to retrieve the resource throws an exception, the [GetResourceData](#) method can help identify the source of the exception by supplying information about the resource's data type. For more information about the string that indicates the resource's data type, see [GetResourceData](#).

The following example illustrates how to use this approach to retrieve resources and to handle any exceptions that are thrown. It programmatically creates a binary .resources file that contains four strings, one Boolean, one integer, and one bitmap. To run the example, do the following:

1. Compile and execute the following source code, which creates a .resources file named ContactResources.resources.

C#

```
using System.Drawing;
using System.Drawing.Imaging;
using System.IO;
using System.Resources;
using System.Runtime.Versioning;

public class Example5
{
    [SupportedOSPlatform("windows")]
    public static void Run()
    {
        // Bitmap as stream.
        MemoryStream bitmapStream = new MemoryStream();
        Bitmap bmp = new Bitmap(@".\ContactsIcon.jpg");
        bmp.Save(bitmapStream, ImageFormat.Jpeg);

        // Define resources to be written.
        using (ResourceWriter rw = new
        ResourceWriter(@".\ContactResources.resources"))
        {
            rw.AddResource("Title", "Contact List");
            rw.AddResource("NColumns", 5);
            rw.AddResource("Icon", bitmapStream);
            rw.AddResource("Header1", "Name");
            rw.AddResource("Header2", "City");
            rw.AddResource("Header3", "State");
            rw.AddResource("ClientVersion", true);
            rw.Generate();
        }
    }
}
```

The source code file is named CreateResources.cs. You can compile it in C# by using the following command:

```
csc CreateResources.cs /r:library.dll
```

2. Compile and run the following code to enumerate the resources in the ContactResources.resources file.

```
C#  
  
using System;  
using System.Collections;  
using System.Drawing;  
using System.IO;  
using System.Resources;  
using System.Runtime.Versioning;  
  
public class Example6  
{  
    [SupportedOSPlatform("windows")]  
    public static void Run()  
    {  
        ResourceReader rdr = new  
ResourceReader(@".\ContactResources.resources");  
        IDictionaryEnumerator dict = rdr.Getenumerator();  
        while (dict.MoveNext())  
        {  
            Console.WriteLine("Resource Name: {0}", dict.Key);  
            try  
            {  
                Console.WriteLine("    Value: {0}", dict.Value);  
            }  
            catch (FileNotFoundException)  
            {  
                Console.WriteLine("    Exception: A file cannot be  
found.");  
                DisplayResourceInfo(rdr, (string)dict.Key, false);  
            }  
            catch (FormatException)  
            {  
                Console.WriteLine("    Exception: Corrupted data.");  
                DisplayResourceInfo(rdr, (string)dict.Key, true);  
            }  
            catch (TypeLoadException)  
            {  
                Console.WriteLine("    Exception: Cannot load the data  
type.");  
                DisplayResourceInfo(rdr, (string)dict.Key, false);  
            }  
        }  
    }  
  
    [SupportedOSPlatform("windows")]  
    private static void DisplayResourceInfo(ResourceReader rr,  
                                            string key, bool loaded)  
    {  
        string dataType = null;  
        byte[] data = null;  
        rr.GetResourceData(key, out dataType, out data);
```

```

// Display the data type.
Console.WriteLine("    Data Type: {0}", dataType);
// Display the bytes that form the available data.
Console.Write("    Data: ");
int lines = 0;
foreach (var dataItem in data)
{
    lines++;
    Console.Write("{0:X2} ", dataItem);
    if (lines % 25 == 0)
        Console.WriteLine("\n");
}
Console.WriteLine();
// Try to recreate current state of data.
// Do: Bitmap, DateTimeTzI
switch (dataType)
{
    // Handle internally serialized string data
    // (ResourceTypeCode members).
    case "ResourceTypeCode.String":
        BinaryReader reader = new BinaryReader(new
MemoryStream(data));
        string binData = reader.ReadString();
        Console.WriteLine("    Recreated Value: {0}", binData);
        break;
    case "ResourceTypeCode.Int32":
        Console.WriteLine("    Recreated Value: {0}",
BitConverter.ToInt32(data, 0));
        break;
    case "ResourceTypeCode.Boolean":
        Console.WriteLine("    Recreated Value: {0}",
BitConverter.ToBoolean(data, 0));
        break;
    // .jpeg image stored as a stream.
    case "ResourceTypeCode.Stream":
        const int OFFSET = 4;
        int size = BitConverter.ToInt32(data, 0);
        Bitmap value1 = new Bitmap(new MemoryStream(data,
OFFSET, size));
        Console.WriteLine("    Recreated Value: {0}", value1);
        break;
    default:
        break;
}
Console.WriteLine();
}
}

```

After modifying the source code (for example, by deliberately throwing a `FormatException` at the end of the `try` block), you can run the example to see how

calls to [GetResourceData](#) enable you to retrieve or recreate some resource information.

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

.NET is an open source project.
Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

System.Resources.SatelliteContractVersionAttribute class

Article • 01/08/2024

This article provides supplementary remarks to the reference documentation for this API.

In desktop apps, the [SatelliteContractVersionAttribute](#) attribute establishes a contract between a main assembly and all its satellites. You apply this attribute to your main assembly, and pass it the version number of the satellite assemblies that will work with this version of the main assembly. When the resource manager ([ResourceManager](#) object) looks up resources, it explicitly loads the satellite version specified by this attribute on the main assembly.

When you update the main assembly, you increment its assembly version number. However, you might not want to ship new copies of your satellite assemblies if the existing ones are compatible with your app. In this case, increment the main assembly's version number but leave the satellite contract version number the same. The resource manager will use your existing satellite assemblies.

If you want to revise a satellite assembly but not the main assembly, increment the version number on your satellite. In this case, ship a publisher policy assembly along with your satellite assembly stating that your new satellite assembly has backward compatibility with your old satellite assembly. The resource manager will still use the old contract number written into your main assembly based on the [SatelliteContractVersionAttribute](#) attribute; however, the loader will bind to the satellite assembly version that is specified by the policy assembly.

A vendor of a shared component uses a publisher policy assembly to make a compatibility statement about a particular version of a released assembly. A publisher policy assembly is a strongly named assembly that has a name in the format `policy.<major>.<minor>.<ComponentAssemblyName>`, and is registered in the [Global Assembly Cache \(GAC\)](#). The publisher policy is generated from an XML configuration file (see the [<bindingRedirect> Element](#)) by using the [Al.exe \(Assembly Linker\)](#) tool. The Assembly Linker is used with the `/link` option to link the XML configuration file to a manifest assembly, which is then stored in the global assembly cache. The publisher policy assemblies can be used when a vendor ships a maintenance release (service pack) that contains bug fixes.

Windows 8.x Store apps

This attribute is ignored in Windows 8.x Store apps, because package resource index (PRI) files do not have versioning semantics. In addition, the Windows 8.x Store packaging model requires all resources to ship in the same package, with no possibility of redeploying satellite assemblies or PRI files.

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

.NET is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

Worker services in .NET

Article • 12/15/2023

There are numerous reasons for creating long-running services such as:

- Processing CPU-intensive data.
- Queuing work items in the background.
- Performing a time-based operation on a schedule.

Background service processing usually doesn't involve a user interface (UI), but UIs can be built around them. In the early days with .NET Framework, Windows developers could create Windows Services for these reasons. Now with .NET, you can use the [BackgroundService](#), which is an implementation of [IHostedService](#), or implement your own.

With .NET, you're no longer restricted to Windows. You can develop cross-platform background services. Hosted services are logging, configuration, and dependency injection (DI) ready. They're a part of the extensions suite of libraries, meaning they're fundamental to all .NET workloads that work with the [generic host](#).

ⓘ Important

Installing the .NET SDK also installs the `Microsoft.NET.Sdk.Worker` and the worker template. In other words, after installing the .NET SDK, you could create a new worker by using the `dotnet new worker` command. If you're using Visual Studio, the template is hidden until the optional ASP.NET and web development workload is installed.

Terminology

Many terms are mistakenly used synonymously. In this section, there are definitions for some of these terms to make their intent more apparent.

- **Background Service:** Refers to the [BackgroundService](#) type.
- **Hosted Service:** Implementations of [IHostedService](#), or referring to the [IHostedService](#) itself.
- **Long-running Service:** Any service that runs continuously.
- **Windows Service:** The *Windows Service* infrastructure, originally .NET Framework centric but now accessible via .NET.
- **Worker Service:** Refers to the *Worker Service* template.

Worker Service template

The Worker Service template is available to the .NET CLI, and Visual Studio. For more information, see [.NET CLI, dotnet new worker - template](#). The template consists of a `Program` and `Worker` class.

C#

```
using App.WorkerService;

HostApplicationBuilder builder = Host.CreateApplicationBuilder(args);
builder.Services.AddHostedService<Worker>();

IHost host = builder.Build();
host.Run();
```

The preceding `Program` class:

- Creates a `HostApplicationBuilder`.
- Calls `AddHostedService` to register the `Worker` as a hosted service.
- Builds an `IHost` from the builder.
- Calls `Run` on the `host` instance, which runs the app.

Template defaults

The Worker template doesn't enable server garbage collection (GC) by default, as there are numerous factors that play a role in determining its necessity. All of the scenarios that require long-running services should consider performance implications of this default. To enable server GC, add the `ServerGarbageCollection` node to the project file:

XML

```
<PropertyGroup>
    <ServerGarbageCollection>true</ServerGarbageCollection>
</PropertyGroup>
```

Tradeoffs and considerations

 Expand table

| Enabled | Disabled |
|---|--|
| Efficient memory management: Automatically reclaims unused memory to prevent memory | Improved real-time performance: Avoids potential pauses or interruptions caused by |

| Enabled | Disabled |
|---|--|
| leaks and optimize resource usage. | garbage collection in latency-sensitive applications. |
| Long-term stability: Helps maintain stable performance in long-running services by managing memory over extended periods. | Resource efficiency: May conserve CPU and memory resources in resource-constrained environments. |
| Reduced maintenance: Minimizes the need for manual memory management, simplifying maintenance. | Manual memory control: Provides fine-grained control over memory for specialized applications. |
| Predictable behavior: Contributes to consistent and predictable application behavior. | Suitable for Short-lived processes: Minimizes the overhead of garbage collection for short-lived or ephemeral processes. |

For more information regarding performance considerations, see [Server GC](#). For more information on configuring server GC, see [Server GC configuration examples](#).

Worker class

As for the `Worker`, the template provides a simple implementation.

```
C#  
  
namespace App.WorkerService;  
  
public sealed class Worker(ILogger<Worker> logger) : BackgroundService  
{  
    protected override async Task ExecuteAsync(CancellationToken stoppingToken)  
    {  
        while (!stoppingToken.IsCancellationRequested)  
        {  
            logger.LogInformation("Worker running at: {time}",  
DateTimeOffset.Now);  
            await Task.Delay(1_000, stoppingToken);  
        }  
    }  
}
```

The preceding `Worker` class is a subclass of [BackgroundService](#), which implements [IHostedService](#). The [BackgroundService](#) is an `abstract class` and requires the subclass to implement `BackgroundService.ExecuteAsync(CancellationToken)`. In the template implementation, the `ExecuteAsync` loops once per second, logging the current date and time until the process is signaled to cancel.

The project file

The Worker template relies on the following project file `Sdk`:

XML

```
<Project Sdk="Microsoft.NET.Sdk.Worker">
```

For more information, see [.NET project SDKs](#).

NuGet package

An app based on the Worker template uses the `Microsoft.NET.Sdk.Worker` SDK and has an explicit package reference to the [Microsoft.Extensions.Hosting](#) package.

Containers and cloud adaptability

With most modern .NET workloads, containers are a viable option. When creating a long-running service from the Worker template in Visual Studio, you can opt in to **Docker support**. Doing so creates a *Dockerfile* that containerizes your .NET app. A [Dockerfile](#) is a set of instructions to build an image. For .NET apps, the *Dockerfile* usually sits in the root of the directory next to a solution file.

Dockerfile

```
# See https://aka.ms/containerfastmode to understand how Visual Studio uses
# this
# Dockerfile to build your images for faster debugging.

FROM mcr.microsoft.com/dotnet/runtime:8.0 AS base
WORKDIR /app

FROM mcr.microsoft.com/dotnet/sdk:8.0 AS build
WORKDIR /src
COPY ["background-service/App.WorkerService.csproj", "background-service/"]
RUN dotnet restore "background-service/App.WorkerService.csproj"
COPY .
WORKDIR "/src/background-service"
RUN dotnet build "App.WorkerService.csproj" -c Release -o /app/build

FROM build AS publish
RUN dotnet publish "App.WorkerService.csproj" -c Release -o /app/publish

FROM base AS final
WORKDIR /app
```

```
COPY --from=publish /app/publish .
ENTRYPOINT ["dotnet", "App.WorkerService.dll"]
```

The preceding *Dockerfile* steps include:

- Setting the base image from `mcr.microsoft.com/dotnet/runtime:8.0` as the alias `base`.
- Changing the working directory to `/app`.
- Setting the `build` alias from the `mcr.microsoft.com/dotnet/sdk:8.0` image.
- Changing the working directory to `/src`.
- Copying the contents and publishing the .NET app:
 - The app is published using the `dotnet publish` command.
- Relayering the .NET SDK image from `mcr.microsoft.com/dotnet/runtime:8.0` (the `base` alias).
- Copying the published build output from the `/publish`.
- Defining the entry point, which delegates to `dotnet App.BackgroundService.dll`.

💡 Tip

The MCR in `mcr.microsoft.com` stands for "Microsoft Container Registry", and is Microsoft's syndicated container catalog from the official Docker hub. The [Microsoft syndicates container catalog](#) article contains additional details.

When you target Docker as a deployment strategy for your .NET Worker Service, there are a few considerations in the project file:

XML

```
<Project Sdk="Microsoft.NET.Sdk.Worker">

<PropertyGroup>
    <TargetFramework>net8.0</TargetFramework>
    <Nullable>enable</Nullable>
    <ImplicitUsings>true</ImplicitUsings>
    <RootNamespace>App.WorkerService</RootNamespace>
    <DockerDefaultTargetOS>Linux</DockerDefaultTargetOS>
</PropertyGroup>

<ItemGroup>
    <PackageReference Include="Microsoft.Extensions.Hosting" Version="8.0.0" />
    <PackageReference
        Include="Microsoft.VisualStudio.Azure.Containers.Tools.Targets"
        Version="1.19.5" />
</ItemGroup>
```

```
</ItemGroup>  
</Project>
```

In the preceding project file, the `<DockerDefaultTargetOS>` element specifies `Linux` as its target. To target Windows containers, use `Windows` instead. The [Microsoft.VisualStudio.Azure.Containers.Tools.Targets NuGet package](#) is automatically added as a package reference when **Docker support** is selected from the template.

For more information on Docker with .NET, see [Tutorial: Containerize a .NET app](#). For more information on deploying to Azure, see [Tutorial: Deploy a Worker Service to Azure](#).

ⓘ Important

If you want to leverage *User Secrets* with the Worker template, you'd have to explicitly reference the `Microsoft.Extensions.Configuration.UserSecrets` NuGet package.

Hosted Service extensibility

The [IHostedService](#) interface defines two methods:

- [IHostedService.StartAsync\(CancellationToken\)](#)
- [IHostedService.StopAsync\(CancellationToken\)](#)

These two methods serve as *lifecycle* methods - they're called during host start and stop events respectively.

ⓘ Note

When overriding either `StartAsync` or `StopAsync` methods, you must call and `await` the `base` class method to ensure the service starts and/or shuts down properly.

ⓘ Important

The interface serves as a generic-type parameter constraint on the [AddHostedService<THostedService>\(IServiceCollection\)](#) extension method, meaning only implementations are permitted. You're free to use the provided [BackgroundService](#) with a subclass, or implement your own entirely.

Signal completion

In most common scenarios, you don't need to explicitly signal the completion of a hosted service. When the host starts the services, they're designed to run until the host is stopped. In some scenarios, however, you may need to signal the completion of the entire host application when the service completes. To signal the completion, consider the following `Worker` class:

C#

```
namespace App.SignalCompletionService;

public sealed class Worker(
    IHostApplicationLifetime hostApplicationLifetime,
    ILogger<Worker> logger) : BackgroundService
{
    protected override async Task ExecuteAsync(CancellationToken
stoppingToken)
    {
        // TODO: implement single execution logic here.
        logger.LogInformation(
            "Worker running at: {Time}", DateTimeOffset.Now);

        await Task.Delay(1_000, stoppingToken);

        // When completed, the entire app host will stop.
        hostApplicationLifetime.StopApplication();
    }
}
```

In the preceding code, the `ExecuteAsync` method doesn't loop, and when it's complete it calls `IHostApplicationLifetime.StopApplication()`.

ⓘ Important

This will signal to the host that it should stop, and without this call to `StopApplication` the host will continue to run indefinitely.

For more information, see:

- [.NET Generic Host: IHostApplicationLifetime](#)
- [.NET Generic Host: Host shutdown](#)
- [.NET Generic Host: Hosting shutdown process](#)

See also

- [BackgroundService](#) subclass tutorials:
 - [Create a Queue Service in .NET](#)
 - [Use scoped services within a BackgroundService in .NET](#)
 - [Create a Windows Service using BackgroundService in .NET](#)
- Custom [IHostedService](#) implementation:
 - [Implement the IHostedService interface in .NET](#)

 [Collaborate with us on GitHub](#)

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

.NET is an open source project.
Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

Create a Queue Service

Article • 12/15/2023

A queue service is a great example of a long-running service, where work items can be queued and worked on sequentially as previous work items are completed. Relying on the Worker Service template, you build out new functionality on top of the [BackgroundService](#).

In this tutorial, you learn how to:

- ✓ Create a queue service.
- ✓ Delegate work to a task queue.
- ✓ Register a console key-listener from [IHostApplicationLifetime](#) events.

💡 Tip

All of the "Workers in .NET" example source code is available in the [Samples Browser](#) for download. For more information, see [Browse code samples: Workers in .NET](#).

Prerequisites

- The [.NET 8.0 SDK or later](#)
- A .NET integrated development environment (IDE)
 - Feel free to use [Visual Studio](#)

Create a new project

To create a new Worker Service project with Visual Studio, you'd select **File > New > Project....** From the **Create a new project** dialog search for "Worker Service", and select Worker Service template. If you'd rather use the .NET CLI, open your favorite terminal in a working directory. Run the `dotnet new` command, and replace the `<Project.Name>` with your desired project name.

.NET CLI

```
dotnet new worker --name <Project.Name>
```

For more information on the .NET CLI new worker service project command, see [dotnet new worker](#).

💡 Tip

If you're using Visual Studio Code, you can run .NET CLI commands from the integrated terminal. For more information, see [Visual Studio Code: Integrated Terminal](#).

Create queuing services

You may be familiar with the

`QueueBackgroundWorkItem(Func<CancellationToken, Task>)` functionality from the `System.Web.Hosting` namespace.

💡 Tip

The functionality of the `System.Web` namespace was intentionally not ported over to .NET, and remains exclusive to .NET Framework. For more information, see [Get started with incremental ASP.NET to ASP.NET Core migration](#).

In .NET, to model a service that is inspired by the `QueueBackgroundWorkItem` functionality, start by adding an `IBackgroundTaskQueue` interface to the project:

C#

```
namespace App.QueueService;

public interface IBackgroundTaskQueue
{
    ValueTask QueueBackgroundWorkItemAsync(
        Func<CancellationToken, ValueTask> workItem);

    ValueTask<Func<CancellationToken, ValueTask>> DequeueAsync(
        CancellationToken cancellationToken);
}
```

There are two methods, one that exposes queuing functionality, and another that dequeues previously queued work items. A *work item* is a `Func<CancellationToken, ValueTask>`. Next, add the default implementation to the project.

C#

```

using System.Threading.Channels;

namespace App.QueueService;

public sealed class DefaultBackgroundTaskQueue : IBackgroundTaskQueue
{
    private readonly Channel<Func< CancellationToken, ValueTask>> _queue;

    public DefaultBackgroundTaskQueue(int capacity)
    {
        BoundedChannelOptions options = new(capacity)
        {
            FullMode = BoundedChannelFullMode.Wait
        };
        _queue = Channel.CreateBounded<Func< CancellationToken, ValueTask>>
(options);
    }

    public async ValueTask QueueBackgroundWorkItemAsync(
        Func< CancellationToken, ValueTask> workItem)
    {
        ArgumentNullException.ThrowIfNull(workItem);

        await _queue.Writer.WriteAsync(workItem);
    }

    public async ValueTask<Func< CancellationToken, ValueTask>> DequeueAsync(
        CancellationToken cancellationToken)
    {
        Func< CancellationToken, ValueTask>? workItem =
            await _queue.Reader.ReadAsync(cancellationToken);

        return workItem;
    }
}

```

The preceding implementation relies on a [Channel<T>](#) as a queue. The [BoundedChannelOptions\(Int32\)](#) is called with an explicit capacity. Capacity should be set based on the expected application load and number of concurrent threads accessing the queue. [BoundedChannelFullMode.Wait](#) causes calls to [ChannelWriter<T>.WriteAsync](#) to return a task, which completes only when space becomes available. Which leads to backpressure, in case too many publishers/calls start accumulating.

Rewrite the Worker class

In the following [QueueHostedService](#) example:

- The `ProcessTaskQueueAsync` method returns a `Task` in `ExecuteAsync`.
- Background tasks in the queue are dequeued and executed in `ProcessTaskQueueAsync`.
- Work items are awaited before the service stops in `StopAsync`.

Replace the existing `Worker` class with the following C# code, and rename the file to `QueueHostedService.cs`.

C#

```
namespace App.QueueService;

public sealed class QueuedHostedService(
    IBackgroundTaskQueue taskQueue,
    ILogger<QueuedHostedService> logger) : BackgroundService
{
    protected override Task ExecuteAsync(CancellationToken stoppingToken)
    {
        logger.LogInformation(
            "{Name} is running.
            Tap W to add a work item to the
            background queue.
            ",
            nameof(QueuedHostedService));

        return ProcessTaskQueueAsync(stoppingToken);
    }

    private async Task ProcessTaskQueueAsync(CancellationToken
stoppingToken)
    {
        while (!stoppingToken.IsCancellationRequested)
        {
            try
            {
                Func< CancellationToken, ValueTask>? workItem =
                    await taskQueue.DequeueAsync(stoppingToken);

                await workItem(stoppingToken);
            }
            catch (OperationCanceledException)
            {
                // Prevent throwing if stoppingToken was signaled
            }
            catch (Exception ex)
            {
                logger.LogError(ex, "Error occurred executing task work
item.");
            }
        }
    }
}
```

```
public override async Task StopAsync(CancellationToken stoppingToken)
{
    logger.LogInformation(
        $"{nameof(QueuedHostedService)} is stopping.");

    await base.StopAsync(stoppingToken);
}
}
```

A `MonitorLoop` service handles enqueueing tasks for the hosted service whenever the `w` key is selected on an input device:

- The `IBackgroundTaskQueue` is injected into the `MonitorLoop` service.
- `IBackgroundTaskQueue.QueueBackgroundWorkItemAsync` is called to enqueue a work item.
- The work item simulates a long-running background task:
 - Three 5-second delays are executed `Delay`.
 - A `try-catch` statement traps `OperationCanceledException` if the task is canceled.

C#

```
namespace App.QueueService;

public sealed class MonitorLoop(
    IBackgroundTaskQueue taskQueue,
    ILogger<MonitorLoop> logger,
    IHostApplicationLifetime applicationLifetime)
{
    private readonly CancellationToken _cancellationToken =
        applicationLifetime.ApplicationStopping;

    public void StartMonitorLoop()
    {
        logger.LogInformation($"{nameof(MonitorAsync)} loop is starting.");

        // Run a console user input loop in a background thread
        Task.Run(async () => await MonitorAsync());
    }

    private async ValueTask MonitorAsync()
    {
        while (!_cancellationToken.IsCancellationRequested)
        {
            var keyStroke = Console.ReadKey();
            if (keyStroke.Key == ConsoleKey.W)
            {
                // Enqueue a background work item
                await
                    taskQueue.QueueBackgroundWorkItemAsync(BuildWorkItemAsync);
            }
        }
    }
}
```

```

        }
    }

private async ValueTask BuildWorkItemAsync(CancellationToken token)
{
    // Simulate three 5-second tasks to complete
    // for each enqueued work item

    int delayLoop = 0;
    var guid = Guid.NewGuid();

    logger.LogInformation("Queued work item {Guid} is starting.", guid);

    while (!token.IsCancellationRequested && delayLoop < 3)
    {
        try
        {
            await Task.Delay(TimeSpan.FromSeconds(5), token);
        }
        catch (OperationCanceledException)
        {
            // Prevent throwing if the Delay is cancelled
        }

        ++ delayLoop;

        logger.LogInformation("Queued work item {Guid} is running.
{DelayLoop}/3", guid, delayLoop);
    }

    if (delayLoop is 3)
    {
        logger.LogInformation("Queued Background Task {Guid} is
complete.", guid);
    }
    else
    {
        logger.LogInformation("Queued Background Task {Guid} was
cancelled.", guid);
    }
}
}

```

Replace the existing `Program` contents with the following C# code:

C#

```

using App.QueueService;

HostApplicationBuilder builder = Host.CreateApplicationBuilder(args);

```

```
builder.Services.AddSingleton<MonitorLoop>();
builder.Services.AddHostedService<QueuedHostedService>();
builder.Services.AddSingleton<IBackgroundTaskQueue>(_ =>
{
    if (!int.TryParse(builder.Configuration["QueueCapacity"], out var
queueCapacity))
    {
        queueCapacity = 100;
    }

    return new DefaultBackgroundTaskQueue(queueCapacity);
});

IHost host = builder.Build();

MonitorLoop monitorLoop = host.Services.GetRequiredService<MonitorLoop>()!;
monitorLoop.StartMonitorLoop();

host.Run();
```

The services are registered in *(Program.cs)*. The hosted service is registered with the `AddHostedService` extension method. `MonitorLoop` is started in *Program.cs* top-level statement:

C#

```
MonitorLoop monitorLoop = host.Services.GetRequiredService<MonitorLoop>()!;
monitorLoop.StartMonitorLoop();
```

For more information on registering services, see [Dependency injection in .NET](#).

Verify service functionality

To run the application from Visual Studio, select `F5` or select the **Debug > Start Debugging** menu option. If you're using the .NET CLI, run the `dotnet run` command from the working directory:

.NET CLI

```
dotnet run
```

For more information on the .NET CLI run command, see [dotnet run](#).

When prompted enter the `w` (or `w`) at least once to queue an emulated work item, as shown in the example output:

Output

```
info: App.QueueService.MonitorLoop[0]
    MonitorAsync loop is starting.
info: App.QueueService.QueuedHostedService[0]
    QueuedHostedService is running.

    Tap W to add a work item to the background queue.

info: Microsoft.Hosting.Lifetime[0]
    Application started. Press Ctrl+C to shut down.
info: Microsoft.Hosting.Lifetime[0]
    Hosting environment: Development
info: Microsoft.Hosting.Lifetime[0]
    Content root path: .\queue-service
winfo: App.QueueService.MonitorLoop[0]
    Queued work item 8453f845-ea4a-4bcb-b26e-c76c0d89303e is starting.
info: App.QueueService.MonitorLoop[0]
    Queued work item 8453f845-ea4a-4bcb-b26e-c76c0d89303e is running. 1/3
info: App.QueueService.MonitorLoop[0]
    Queued work item 8453f845-ea4a-4bcb-b26e-c76c0d89303e is running. 2/3
info: App.QueueService.MonitorLoop[0]
    Queued work item 8453f845-ea4a-4bcb-b26e-c76c0d89303e is running. 3/3
info: App.QueueService.MonitorLoop[0]
    Queued Background Task 8453f845-ea4a-4bcb-b26e-c76c0d89303e is
complete.
info: Microsoft.Hosting.Lifetime[0]
    Application is shutting down...
info: App.QueueService.QueuedHostedService[0]
    QueuedHostedService is stopping.
```

If running the application from within Visual Studio, select **Debug > Stop Debugging....**. Alternatively, select **ctrl + c** from the console window to signal cancellation.

See also

- [Worker Services in .NET](#)
- [Use scoped services within a BackgroundService](#)
- [Create a Windows Service using BackgroundService](#)
- [Implement the IHostedService interface](#)
- [Web-Queue-Worker architectural style](#)

 Collaborate with us on
GitHub



.NET feedback

.NET is an open source project.
Select a link to provide feedback:

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

 Open a documentation issue

 Provide product feedback

Use scoped services within a `BackgroundService`

Article • 12/13/2023

When you register implementations of [IHostedService](#) using any of the [AddHostedService](#) extension methods - the service is registered as a singleton. There may be scenarios where you'd like to rely on a scoped service. For more information, see [Dependency injection in .NET: Service lifetimes](#).

In this tutorial, you learn how to:

- ✓ Resolve scoped dependencies in a singleton [BackgroundService](#).
- ✓ Delegate work to a scoped service.
- ✓ Implement an `override` of [BackgroundService.StopAsync\(CancellationToken\)](#).

💡 Tip

All of the "Workers in .NET" example source code is available in the [Samples Browser](#) for download. For more information, see [Browse code samples: Workers in .NET](#).

Prerequisites

- The [.NET 8.0 SDK or later](#)
- A .NET integrated development environment (IDE)
 - Feel free to use [Visual Studio](#)

Create a new project

To create a new Worker Service project with Visual Studio, you'd select **File > New > Project....** From the **Create a new project** dialog search for "Worker Service", and select Worker Service template. If you'd rather use the .NET CLI, open your favorite terminal in a working directory. Run the `dotnet new` command, and replace the `<Project.Name>` with your desired project name.

.NET CLI

```
dotnet new worker --name <Project.Name>
```

For more information on the .NET CLI new worker service project command, see [dotnet new worker](#).

💡 Tip

If you're using Visual Studio Code, you can run .NET CLI commands from the integrated terminal. For more information, see [Visual Studio Code: Integrated Terminal](#).

Create scoped services

To use [scoped services](#) within a `BackgroundService`, create a scope. No scope is created for a hosted service by default. The scoped background service contains the background task's logic.

C#

```
namespace App.ScopedService;

public interface IScopedProcessingService
{
    Task DoWorkAsync(CancellationToken stoppingToken);
}
```

The preceding interface defines a single `DoWorkAsync` method. To define the default implementation:

- The service is asynchronous. The `DoWorkAsync` method returns a `Task`. For demonstration purposes, a delay of ten seconds is awaited in the `DoWorkAsync` method.
- An `ILogger` is injected into the service.:

C#

```
namespace App.ScopedService;

public sealed class DefaultScopedProcessingService(
    ILogger<DefaultScopedProcessingService> logger) :
IScopedProcessingService
{
    private int _executionCount;

    public async Task DoWorkAsync(CancellationToken stoppingToken)
    {
        while (!stoppingToken.IsCancellationRequested)
```

```

    {
        ++ _executionCount;

        logger.LogInformation(
            "{ServiceName} working, execution count: {Count}",
            nameof(DefaultScopedProcessingService),
            _executionCount);

        await Task.Delay(10_000, stoppingToken);
    }
}

```

The hosted service creates a scope to resolve the scoped background service to call its `DoworkAsync` method. `DoworkAsync` returns a `Task`, which is awaited in `ExecuteAsync`:

Rewrite the Worker class

Replace the existing `Worker` class with the following C# code, and rename the file to `ScopedBackgroundService.cs`:

C#

```

namespace App.ScopedService;

public sealed class ScopedBackgroundService(
    IServiceScopeFactory serviceScopeFactory,
    ILogger<ScopedBackgroundService> logger) : BackgroundService
{
    private const string ClassName = nameof(ScopedBackgroundService);

    protected override async Task ExecuteAsync(CancellationToken stoppingToken)
    {
        logger.LogInformation(
            "{Name} is running.", ClassName);

        await DoworkAsync(stoppingToken);
    }

    private async Task DoworkAsync(CancellationToken stoppingToken)
    {
        logger.LogInformation(
            "{Name} is working.", ClassName);
    }
}

```

```
        using (IServiceScope scope = serviceScopeFactory.CreateScope())
    {
        IScopedProcessingService scopedProcessingService =
            scope.ServiceProvider.GetRequiredService<IScopedProcessingService>();

        await scopedProcessingService.DoWorkAsync(stoppingToken);
    }
}

public override async Task StopAsync(CancellationToken stoppingToken)
{
    logger.LogInformation(
        "{Name} is stopping.", ClassName);

    await base.StopAsync(stoppingToken);
}
}
```

In the preceding code, an explicit scope is created and the `IScopedProcessingService` implementation is resolved from the dependency injection service scope factory. The resolved service instance is scoped, and its `DoWorkAsync` method is awaited.

Replace the template *Program.cs* file contents with the following C# code:

C#

```
using App.ScopedService;

HostApplicationBuilder builder = Host.CreateApplicationBuilder(args);
builder.Services.AddHostedService<ScopedBackgroundService>();
builder.Services.AddScoped<IScopedProcessingService,
DefaultScopedProcessingService>();

IHost host = builder.Build();
host.Run();
```

The services are registered in (*Program.cs*). The hosted service is registered with the `AddHostedService` extension method.

For more information on registering services, see [Dependency injection in .NET](#).

Verify service functionality

To run the application from Visual Studio, select `F5` or select the **Debug > Start Debugging** menu option. If you're using the .NET CLI, run the `dotnet run` command from the working directory:

.NET CLI

```
dotnet run
```

For more information on the .NET CLI run command, see [dotnet run](#).

Let the application run for a bit to generate several execution count increments. You will see output similar to the following:

Output

```
info: App.ScopedService.ScopedBackgroundService[0]
      ScopedBackgroundService is running.
info: App.ScopedService.ScopedBackgroundService[0]
      ScopedBackgroundService is working.
info: App.ScopedService.DefaultScopedProcessingService[0]
      DefaultScopedProcessingService working, execution count: 1
info: Microsoft.Hosting.Lifetime[0]
      Application started. Press Ctrl+C to shut down.
info: Microsoft.Hosting.Lifetime[0]
      Hosting environment: Development
info: Microsoft.Hosting.Lifetime[0]
      Content root path: .\scoped-service
info: App.ScopedService.DefaultScopedProcessingService[0]
      DefaultScopedProcessingService working, execution count: 2
info: App.ScopedService.DefaultScopedProcessingService[0]
      DefaultScopedProcessingService working, execution count: 3
info: App.ScopedService.DefaultScopedProcessingService[0]
      DefaultScopedProcessingService working, execution count: 4
info: Microsoft.Hosting.Lifetime[0]
      Application is shutting down...
info: App.ScopedService.ScopedBackgroundService[0]
      ScopedBackgroundService is stopping.
```

If running the application from within Visual Studio, select **Debug > Stop Debugging...**

Alternatively, select **Ctrl + C** from the console window to signal cancellation.

See also

- [Worker Services in .NET](#)
- [Create a Queue Service](#)
- [Create a Windows Service using BackgroundService](#)
- [Implement the IHostedService interface](#)

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

.NET is an open source project.
Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

Create Windows Service using `BackgroundService`

Article • 12/13/2023

.NET Framework developers are probably familiar with Windows Service apps. Before .NET Core and .NET 5+, developers who relied on .NET Framework could create Windows Services to perform background tasks or execute long-running processes. This functionality is still available and you can create Worker Services that run as a Windows Service.

In this tutorial, you'll learn how to:

- ✓ Publish a .NET worker app as a single file executable.
- ✓ Create a Windows Service.
- ✓ Create the `BackgroundService` app as a Windows Service.
- ✓ Start and stop the Windows Service.
- ✓ View event logs.
- ✓ Delete the Windows Service.

💡 Tip

All of the "Workers in .NET" example source code is available in the [Samples Browser](#) for download. For more information, see [Browse code samples: Workers in .NET](#).

ⓘ Important

Installing the .NET SDK also installs the `Microsoft.NET.Sdk.Worker` and the worker template. In other words, after installing the .NET SDK, you could create a new worker by using the `dotnet new worker` command. If you're using Visual Studio, the template is hidden until the optional ASP.NET and web development workload is installed.

Prerequisites

- The [.NET 8.0 SDK or later](#)
- A Windows OS
- A .NET integrated development environment (IDE)

- Feel free to use [Visual Studio](#)

Create a new project

To create a new Worker Service project with Visual Studio, you'd select **File > New > Project...**. From the **Create a new project** dialog search for "Worker Service", and select Worker Service template. If you'd rather use the .NET CLI, open your favorite terminal in a working directory. Run the `dotnet new` command, and replace the `<Project.Name>` with your desired project name.

.NET CLI

```
dotnet new worker --name <Project.Name>
```

For more information on the .NET CLI new worker service project command, see [dotnet new worker](#).

💡 Tip

If you're using Visual Studio Code, you can run .NET CLI commands from the integrated terminal. For more information, see [Visual Studio Code: Integrated Terminal](#).

Install NuGet package

To interop with native Windows Services from .NET `IHostedService` implementations, you'll need to install the [Microsoft.Extensions.Hosting.WindowsServices NuGet package](#).

To install this from Visual Studio, use the **Manage NuGet Packages...** dialog. Search for "Microsoft.Extensions.Hosting.WindowsServices", and install it. If you'd rather use the .NET CLI, run the `dotnet add package` command:

.NET CLI

```
dotnet add package Microsoft.Extensions.Hosting.WindowsServices
```

For more information on the .NET CLI add package command, see [dotnet add package](#).

After successfully adding the packages, your project file should now contain the following package references:

XML

```
<ItemGroup>
  <PackageReference Include="Microsoft.Extensions.Hosting" Version="8.0.0"
/>
  <PackageReference Include="Microsoft.Extensions.Hosting.WindowsServices"
Version="8.0.0" />
</ItemGroup>
```

Update project file

This worker project makes use of C#'s [nullable reference types](#). To enable them for the entire project, update the project file accordingly:

XML

```
<Project Sdk="Microsoft.NET.Sdk.Worker">

  <PropertyGroup>
    <TargetFramework>net8.0-windows</TargetFramework>
    <Nullable>enable</Nullable>
    <ImplicitUsings>true</ImplicitUsings>
    <RootNamespace>App.WindowsService</RootNamespace>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.Extensions.Hosting" Version="8.0.0"
/>
    <PackageReference Include="Microsoft.Extensions.Hosting.WindowsServices"
Version="8.0.0" />
  </ItemGroup>
</Project>
```

The preceding project file changes add the `<Nullable>enable</Nullable>` node. For more information, see [Setting the nullable context](#).

Create the service

Add a new class to the project named *JokeService.cs*, and replace its contents with the following C# code:

C#

```
namespace App.WindowsService;

public sealed class JokeService
```

```
{  
    public string GetJoke()  
    {  
        Joke joke = _jokes.ElementAt(  
            Random.Shared.Next(_jokes.Count));  
  
        return $"{joke.Setup}{Environment.NewLine}{joke.Punchline}";  
    }  
  
    // Programming jokes borrowed from:  
    // https://github.com/eklavyadev/karljoke/blob/main/source/jokes.json  
    private readonly HashSet<Joke> _jokes = new()  
    {  
        new Joke("What's the best thing about a Boolean?", "Even if you're  
        wrong, you're only off by a bit."),  
        new Joke("What's the object-oriented way to become wealthy?",  
        "Inheritance"),  
        new Joke("Why did the programmer quit their job?", "Because they  
        didn't get arrays."),  
        new Joke("Why do programmers always mix up Halloween and  
        Christmas?", "Because Oct 31 == Dec 25"),  
        new Joke("How many programmers does it take to change a lightbulb?",  
        "None that's a hardware problem"),  
        new Joke("If you put a million monkeys at a million keyboards, one  
        of them will eventually write a Java program", "the rest of them will write  
        Perl"),  
        new Joke("[ 'hip', 'hip' ]", "(hip hip array)"),  
        new Joke("To understand what recursion is...", "You must first  
        understand what recursion is"),  
        new Joke("There are 10 types of people in this world...", "Those who  
        understand binary and those who don't"),  
        new Joke("Which song would an exception sing?", "Can't catch me -  
        Avicii"),  
        new Joke("Why do Java programmers wear glasses?", "Because they  
        don't C#"),  
        new Joke("How do you check if a webpage is HTML5?", "Try it out on  
        Internet Explorer"),  
        new Joke("A user interface is like a joke.", "If you have to explain  
        it then it is not that good."),  
        new Joke("I was gonna tell you a joke about UDP...", "...but you  
        might not get it."),  
        new Joke("The punchline often arrives before the set-up.", "Do you  
        know the problem with UDP jokes?"),  
        new Joke("Why do C# and Java developers keep breaking their  
        keyboards?", "Because they use a strongly typed language."),  
        new Joke("Knock-knock.", "A race condition. Who is there?"),  
        new Joke("What's the best part about TCP jokes?", "I get to keep  
        telling them until you get them."),  
        new Joke("A programmer puts two glasses on their bedside table  
        before going to sleep.", "A full one, in case they gets thirsty, and an  
        empty one, in case they don't."),  
        new Joke("There are 10 kinds of people in this world.", "Those who  
        understand binary, those who don't, and those who weren't expecting a base 3  
        joke."),  
        new Joke("What did the router say to the doctor?", "It hurts when
```

```
IP."),  
    new Joke("An IPv6 packet is walking out of the house.", "He goes  
nowhere."),  
    new Joke("3 SQL statements walk into a NoSQL bar. Soon, they walk  
out", "They couldn't find a table.")  
};  
}  
  
readonly record struct Joke(string Setup, string Punchline);
```

The preceding joke service source code exposes a single piece of functionality, the `GetJoke` method. This is a `string` returning method that represents a random programming joke. The class-scoped `_jokes` field is used to store the list of jokes. A random joke is selected from the list and returned.

Rewrite the Worker class

Replace the existing `Worker` from the template with the following C# code, and rename the file to *WindowsBackgroundService.cs*:

```
C#  
  
namespace App.WindowsService;  
  
public sealed class WindowsBackgroundService(  
    JokeService jokeService,  
    ILogger<WindowsBackgroundService> logger) : BackgroundService  
{  
    protected override async Task ExecuteAsync(CancellationToken  
stoppingToken)  
    {  
        try  
        {  
            while (!stoppingToken.IsCancellationRequested)  
            {  
                string joke = jokeService.GetJoke();  
                logger.LogWarning("{Joke}", joke);  
  
                await Task.Delay(TimeSpan.FromMinutes(1), stoppingToken);  
            }  
        }  
        catch (OperationCanceledException)  
        {  
            // When the stopping token is canceled, for example, a call made  
from services.msc,  
            // we shouldn't exit with a non-zero exit code. In other words,  
this is expected...  
        }  
        catch (Exception ex)  
        {
```

```

        logger.LogError(ex, "{Message}", ex.Message);

        // Terminates this process and returns an exit code to the
        // operating system.
        // This is required to avoid the
        'BackgroundServiceExceptionBehavior', which
            // performs one of two scenarios:
            // 1. When set to "Ignore": will do nothing at all, errors cause
            zombie services.
            // 2. When set to "StopHost": will cleanly stop the host, and
            log errors.
            //
            // In order for the Windows Service Management system to
            leverage configured
                // recovery options, we need to terminate the process with a
                non-zero exit code.
                Environment.Exit(1);
            }
        }
    }
}

```

In the preceding code, the `JokeService` is injected along with an `ILogger`. Both are made available to the class as `private readonly` fields. In the `ExecuteAsync` method, the joke service requests a joke and writes it to the logger. In this case, the logger is implemented by the Windows Event Log - `Microsoft.Extensions.Logging.EventLog.EventLogLogger`. Logs are written to, and available for viewing in the **Event Viewer**.

① Note

By default, the *Event Log* severity is **Warning**. This can be configured, but for demonstration purposes the `WindowsBackgroundService` logs with the **LogWarning** extension method. To specifically target the `EventLog` level, add an entry in the `appsettings.{Environment}.json`, or provide an `EventLogSettings.Filter` value.

JSON

```
{
    "Logging": {
        "LogLevel": {
            "Default": "Warning"
        },
        "EventLog": {
            "SourceName": "The Joke Service",
            "LogName": "Application",
            "LogLevel": {
                "Microsoft": "Information",
                "Microsoft.Hosting.Lifetime": "Information"
            }
        }
}
```

```
        }  
    }  
}
```

For more information on configuring log levels, see [Logging providers in .NET: Configure Windows EventLog](#).

Rewrite the `Program` class

Replace the template `Program.cs` file contents with the following C# code:

C#

```
using App.WindowsService;  
using Microsoft.Extensions.Logging.Configuration;  
using Microsoft.Extensions.Logging.EventLog;  
  
HostApplicationBuilder builder = Host.CreateApplicationBuilder(args);  
builder.Services.AddWindowsService(options =>  
{  
    options.ServiceName = ".NET Joke Service";  
});  
  
LoggerProviderOptions.RegisterProviderOptions<  
    EventLogSettings, EventLogLoggerProvider>(builder.Services);  
  
builder.Services.AddSingleton<JokeService>();  
builder.Services.AddHostedService<WindowsBackgroundService>();  
  
IHost host = builder.Build();  
host.Run();
```

The `AddWindowsService` extension method configures the app to work as a Windows Service. The service name is set to `".NET Joke Service"`. The hosted service is registered for dependency injection.

For more information on registering services, see [Dependency injection in .NET](#).

Publish the app

To create the .NET Worker Service app as a Windows Service, it's recommended that you publish the app as a single file executable. It's less error-prone to have a self-contained executable, as there aren't any dependent files lying around the file system. But you may choose a different publishing modality, which is perfectly acceptable, so long as you create an `*.exe` file that can be targeted by the Windows Service Control Manager.

ⓘ Important

An alternative publishing approach is to build the *.dll (instead of an *.exe), and when you install the published app using the Windows Service Control Manager you delegate to the .NET CLI and pass the DLL. For more information, see [.NET CLI: dotnet command](#).

PowerShell

```
sc.exe create ".NET Joke Service" binpath="C:\Path\To\dotnet.exe  
C:\Path\To\App.WindowsService.dll"
```

XML

```
<Project Sdk="Microsoft.NET.Sdk.Worker">  
  
  <PropertyGroup>  
    <TargetFramework>net8.0-windows</TargetFramework>  
    <Nullable>enable</Nullable>  
    <ImplicitUsings>true</ImplicitUsings>  
    <RootNamespace>App.WindowsService</RootNamespace>  
    <OutputType>exe</OutputType>  
    <PublishSingleFile Condition="'$(Configuration)' ==  
'Release'">true</PublishSingleFile>  
    <RuntimeIdentifier>win-x64</RuntimeIdentifier>  
    <PlatformTarget>x64</PlatformTarget>  
  </PropertyGroup>  
  
  <ItemGroup>  
    <PackageReference Include="Microsoft.Extensions.Hosting" Version="8.0.0" />  
    <PackageReference Include="Microsoft.Extensions.Hosting.WindowsServices" Version="8.0.0" />  
  </ItemGroup>  
</Project>
```

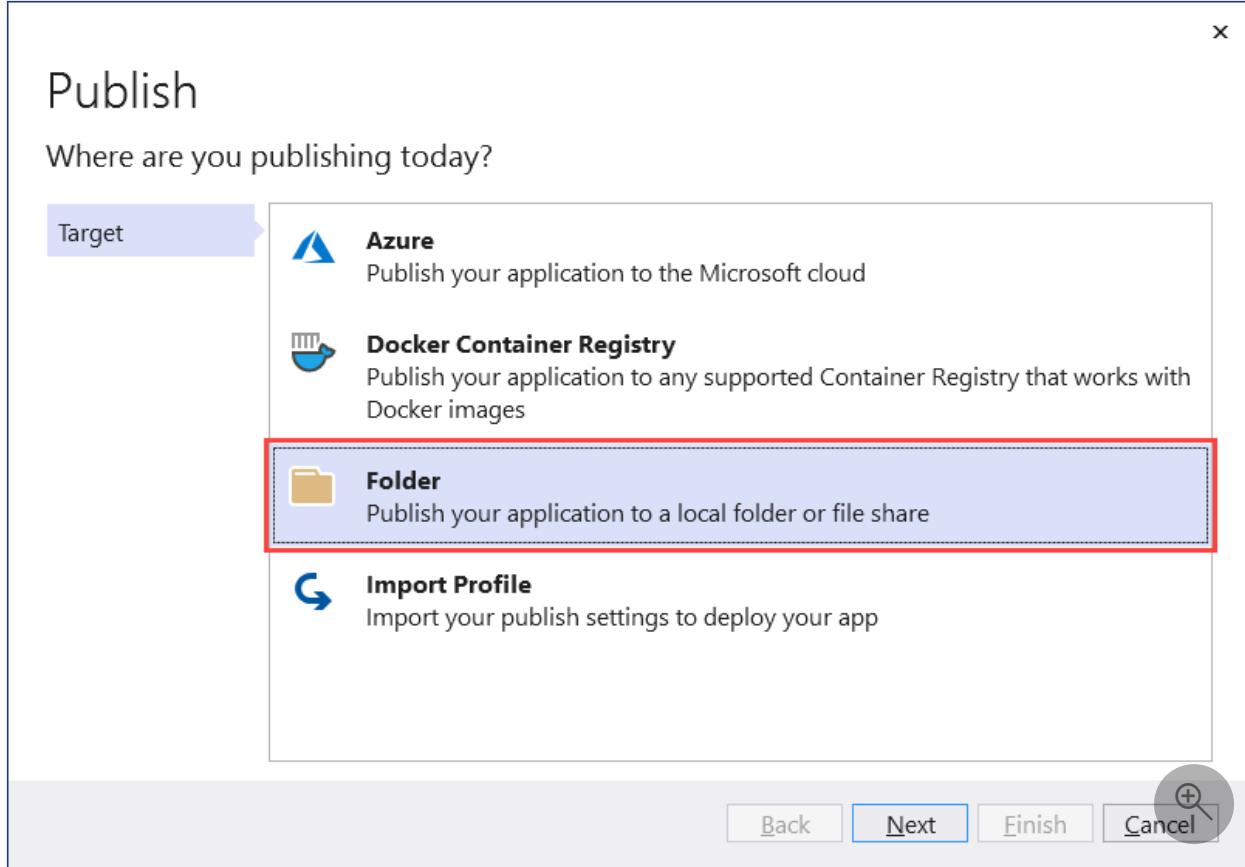
The preceding highlighted lines of the project file define the following behaviors:

- `<OutputType>exe</OutputType>`: Creates a console application.
- `<PublishSingleFile Condition="'$(Configuration)' == 'Release'">true</PublishSingleFile>`: Enables single-file publishing.
- `<RuntimeIdentifier>win-x64</RuntimeIdentifier>`: Specifies the RID of `win-x64`.
- `<PlatformTarget>x64</PlatformTarget>`: Specify the target platform CPU of 64-bit.

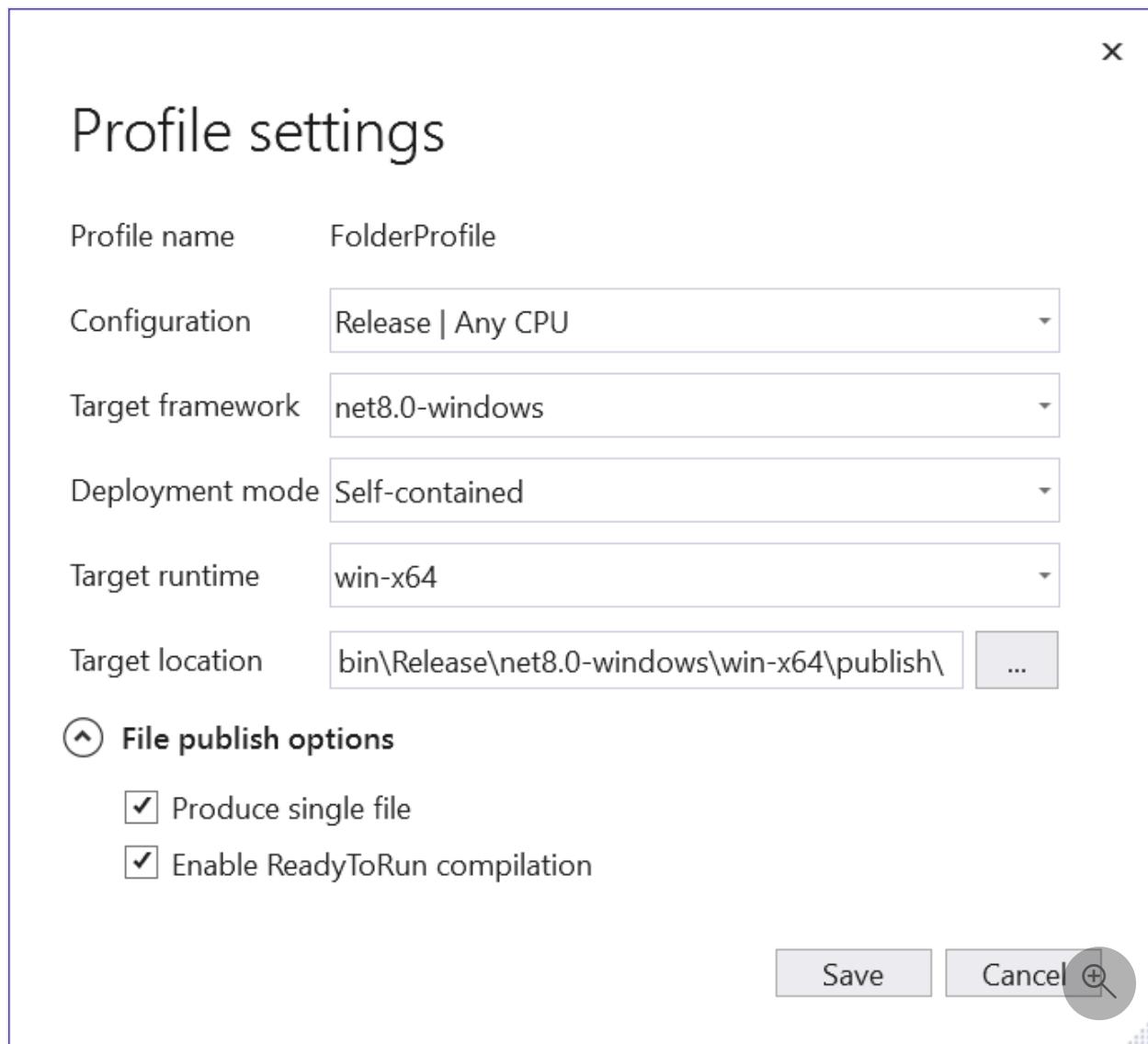
To publish the app from Visual Studio, you can create a publish profile that is persisted. The publish profile is XML-based, and has the `.pubxml` file extension. Visual Studio uses

this profile to publish the app implicitly, whereas if you're using the .NET CLI — you must explicitly specify the publish profile for it to be used.

Right-click on the project in the **Solution Explorer**, and select **Publish....** Then, select **Add a publish profile** to create a profile. From the **Publish** dialog, select **Folder** as your **Target**.



Leave the default **Location**, and then select **Finish**. Once the profile is created, select **Show all settings**, and verify your **Profile settings**.



Ensure that the following settings are specified:

- **Deployment mode:** Self-contained
- **Produce single file:** checked
- **Enable ReadyToRun compilation:** checked
- **Trim unused assemblies (in preview):** unchecked

Finally, select **Publish**. The app is compiled, and the resulting .exe file is published to the */publish* output directory.

Alternatively, you could use the .NET CLI to publish the app:

```
.NET CLI  
  
dotnet publish --output "C:\custom\publish\directory"
```

For more information, see [dotnet publish](#).

Important

With .NET 6, if you attempt to debug the app with the `<PublishSingleFile>true</PublishSingleFile>` setting, you will not be able to debug the app. For more information, see [Unable to attach to CoreCLR when debugging a 'PublishSingleFile' .NET 6 app](#).

Create the Windows Service

If you're unfamiliar with using PowerShell and you'd rather create an installer for your service, see [Create a Windows Service installer](#). Otherwise, to create the Windows Service, use the native Windows Service Control Manager's (`sc.exe`) create command. Run PowerShell as an Administrator.

PowerShell

```
sc.exe create ".NET Joke Service"
binpath="C:\Path\To\App.WindowsService.exe"
```

Tip

If you need to change the content root of the [host configuration](#), you can pass it as a command-line argument when specifying the `binpath`:

PowerShell

```
sc.exe create "Svc Name" binpath="C:\Path\To\App.exe --contentRoot
C:\Other\Path"
```

You'll see an output message:

PowerShell

```
[SC] CreateService SUCCESS
```

For more information, see [sc.exe create](#).

Configure the Windows Service

After the service is created, you can optionally configure it. If you're fine with the service defaults, skip to the [Verify service functionality](#) section.

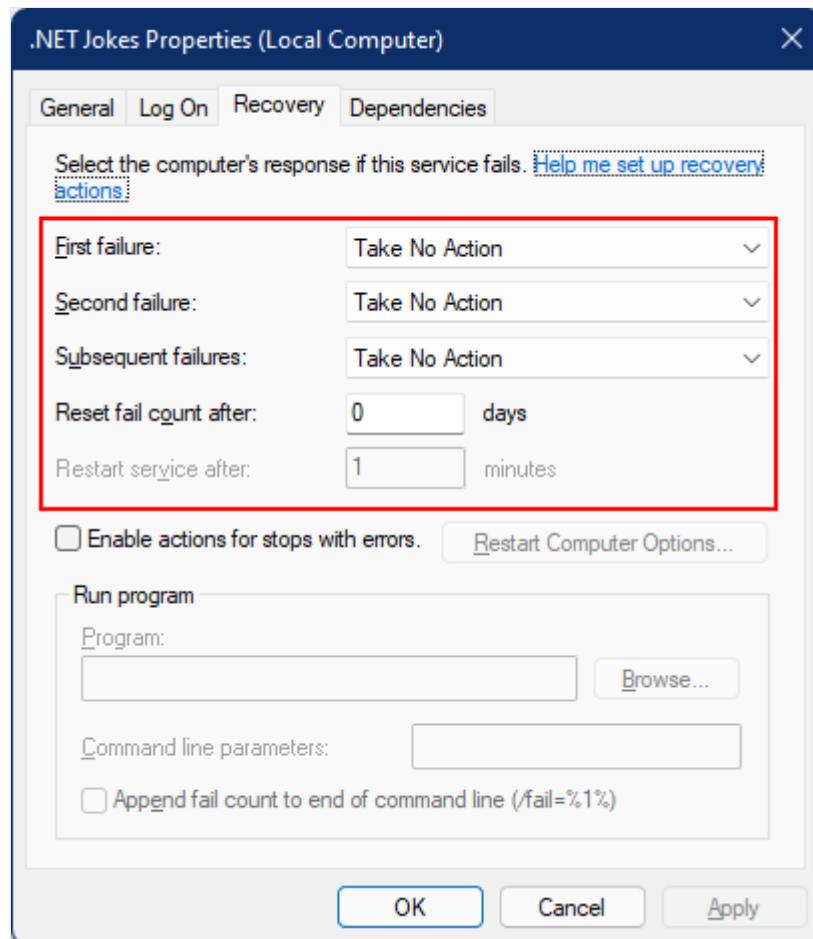
Windows Services provide recovery configuration options. You can query the current configuration using the `sc.exe qfailure "<Service Name>"` (where `<Service Name>` is your services' name) command to read the current recovery configuration values:

```
PowerShell

sc qfailure ".NET Joke Service"
[SC] QueryServiceConfig2 SUCCESS

SERVICE_NAME: .NET Joke Service
    RESET_PERIOD (in seconds)      : 0
    REBOOT_MESSAGE                 :
    COMMAND_LINE                   :
```

The command will output the recovery configuration, which is the default values—since they've not yet been configured.



To configure recovery, use the `sc.exe failure "<Service Name>"` where `<Service Name>` is the name of your service:

```
PowerShell

sc.exe failure ".NET Joke Service" reset=0
actions=restart/60000/restart/60000/run/1000
```

```
[SC] ChangeServiceConfig2 SUCCESS
```

💡 Tip

To configure the recovery options, your terminal session needs to run as an Administrator.

After it's been successfully configured, you can query the values once again using the

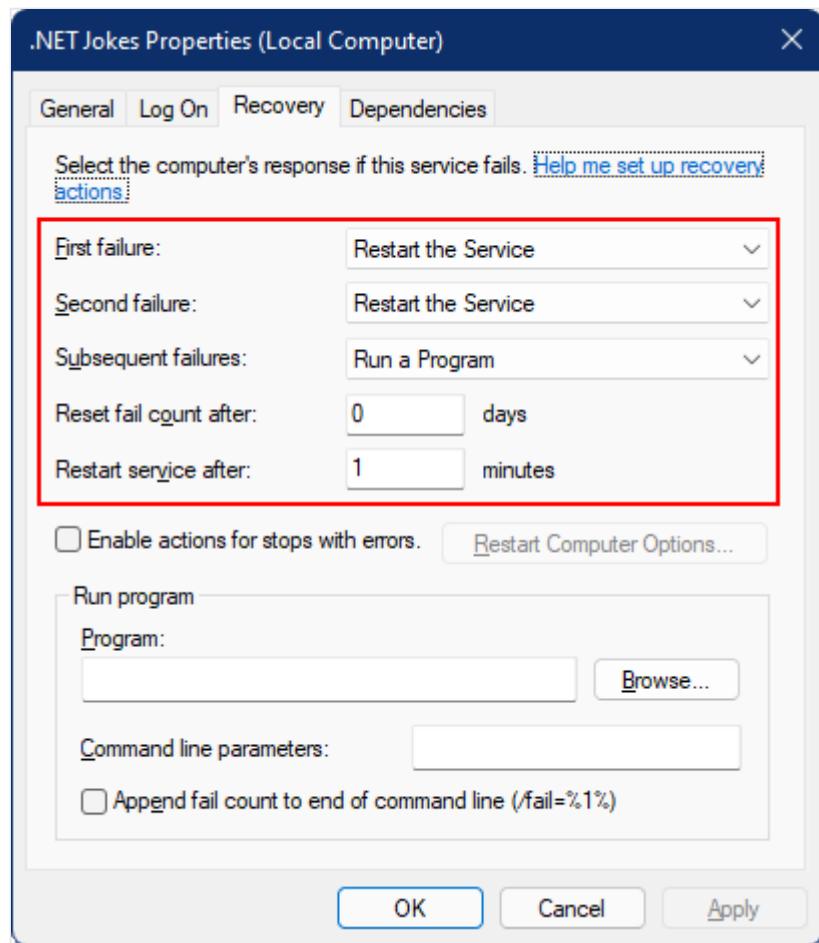
```
sc.exe qfailure "<Service Name>"
```

PowerShell

```
sc qfailure ".NET Joke Service"
[SC] QueryServiceConfig2 SUCCESS

SERVICE_NAME: .NET Joke Service
    RESET_PERIOD (in seconds)      : 0
    REBOOT_MESSAGE                 :
    COMMAND_LINE                   :
    FAILURE_ACTIONS                : RESTART -- Delay = 60000
milliseconds.
                                         RESTART -- Delay = 60000
milliseconds.
                                         RUN PROCESS -- Delay = 1000
milliseconds.
```

You will see the configured restart values.



Service recovery options and .NET `BackgroundService` instances

With .NET 6, [new hosting exception-handling behaviors](#) have been added to .NET. The `BackgroundServiceExceptionBehavior` enum was added to the `Microsoft.Extensions.Hosting` namespace, and is used to specify the behavior of the service when an exception is thrown. The following table lists the available options:

[Expand table](#)

| Option | Description |
|-----------------------|---|
| <code>Ignore</code> | Ignore exceptions thrown in <code>BackgroundService</code> . |
| <code>StopHost</code> | The <code>IHost</code> will be stopped when an unhandled exception is thrown. |

The default behavior before .NET 6 is `Ignore`, which resulted in *zombie processes* (a running process that didn't do anything). With .NET 6, the default behavior is `StopHost`, which results in the host being stopped when an exception is thrown. But it stops cleanly, meaning that the Windows Service management system will not restart the service. To correctly allow the service to be restarted, you can call `Environment.Exit` with a non-zero exit code. Consider the following highlighted `catch` block:

C#

```
namespace App.WindowsService;

public sealed class WindowsBackgroundService(
    JokeService jokeService,
    ILogger<WindowsBackgroundService> logger) : BackgroundService
{
    protected override async Task ExecuteAsync(CancellationToken stoppingToken)
    {
        try
        {
            while (!stoppingToken.IsCancellationRequested)
            {
                string joke = jokeService.GetJoke();
                logger.LogWarning("{Joke}", joke);

                await Task.Delay(TimeSpan.FromMinutes(1), stoppingToken);
            }
        }
        catch (OperationCanceledException)
        {
            // When the stopping token is canceled, for example, a call made
            // from services.msc,
            // we shouldn't exit with a non-zero exit code. In other words,
            // this is expected...
        }
        catch (Exception ex)
        {
            logger.LogError(ex, "{Message}", ex.Message);

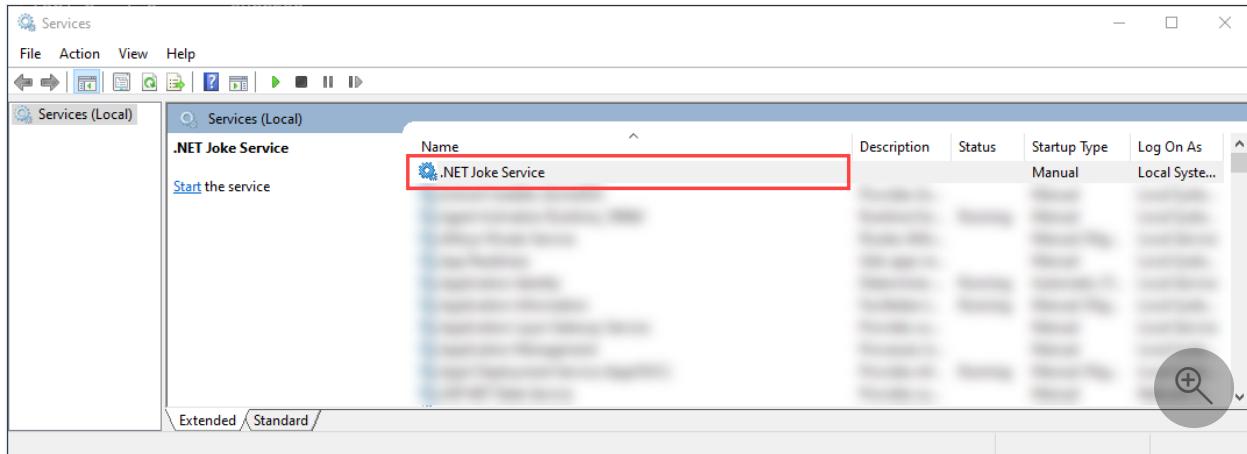
            // Terminates this process and returns an exit code to the
            // operating system.
            // This is required to avoid the
            'BackgroundServiceExceptionBehavior', which
            // performs one of two scenarios:
            // 1. When set to "Ignore": will do nothing at all, errors cause
            // zombie services.
            // 2. When set to "StopHost": will cleanly stop the host, and
            log errors.
            //
            // In order for the Windows Service Management system to
            leverage configured
            // recovery options, we need to terminate the process with a
            non-zero exit code.
            Environment.Exit(1);
        }
    }
}
```

Verify service functionality

To see the app created as a Windows Service, open **Services**. Select the Windows key (or **Ctrl** + **Esc**), and search from "Services". From the **Services** app, you should be able to find your service by its name.

ⓘ Important

By default, regular (non-admin) users cannot manage Windows services. To verify that this app functions as expected, you'll need to use an Admin account.

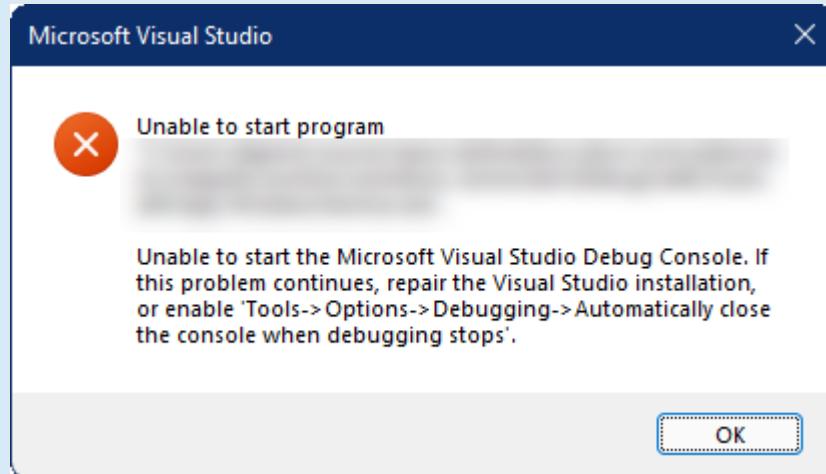


To verify that the service is functioning as expected, you need to:

- Start the service
- View the logs
- Stop the service

ⓘ Important

To debug the application, ensure that you're *not* attempting to debug the executable that is actively running within the Windows Services process.



Start the Windows Service

To start the Windows Service, use the `sc.exe start` command:

```
PowerShell
```

```
sc.exe start ".NET Joke Service"
```

You'll see output similar to the following:

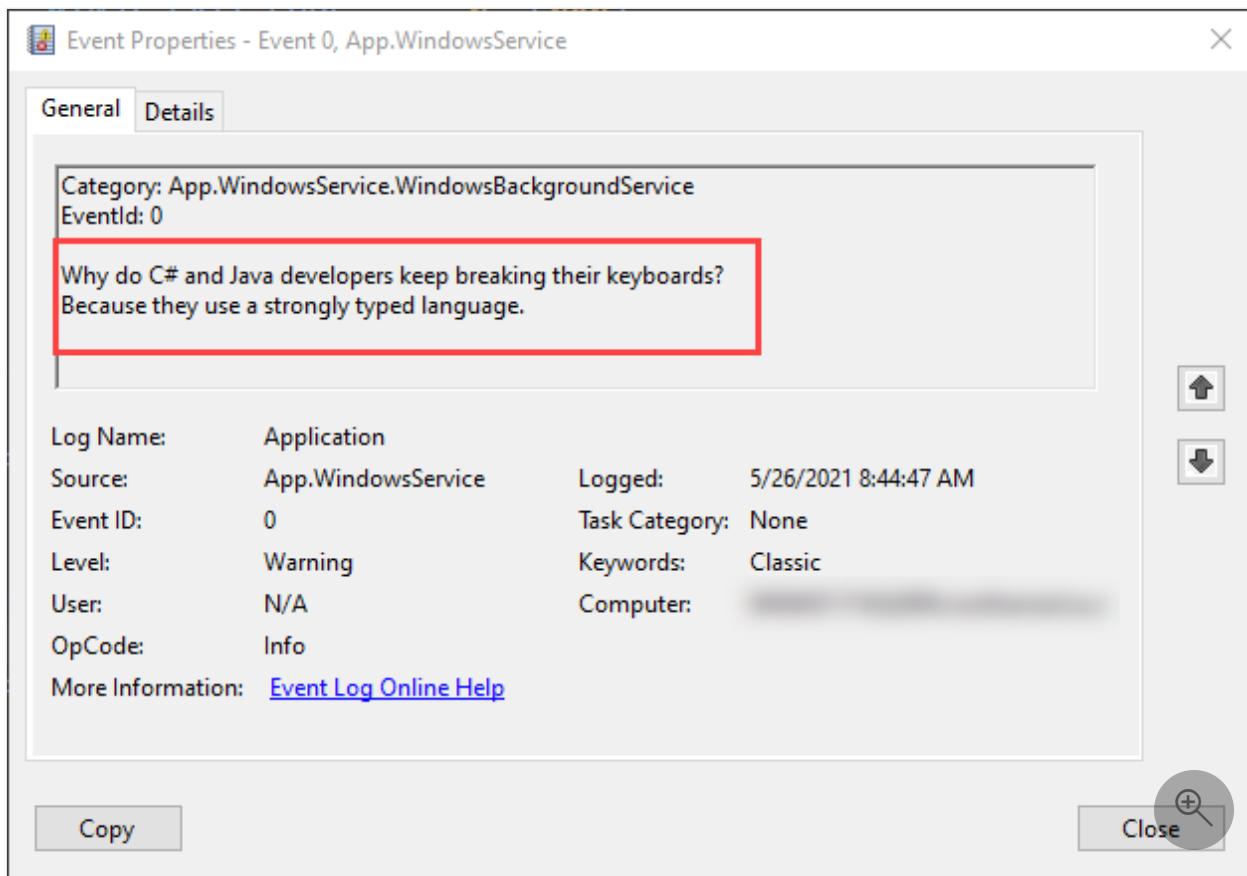
```
PowerShell
```

```
SERVICE_NAME: .NET Joke Service
  TYPE          : 10  WIN32_OWN_PROCESS
  STATE         : 2   START_PENDING
                  (NOT_STOPPABLE, NOT_PAUSABLE, IGNORES_SHUTDOWN)
  WIN32_EXIT_CODE    : 0  (0x0)
  SERVICE_EXIT_CODE  : 0  (0x0)
  CHECKPOINT       : 0x0
  WAIT_HINT        : 0x7d0
  PID              : 37636
  FLAGS
```

The service **Status** will transition out of `START_PENDING` to **Running**.

View logs

To view logs, open the **Event Viewer**. Select the Windows key (or `ctrl` + `Esc`), and search for `"Event Viewer"`. Select the **Event Viewer (Local) > Windows Logs > Application** node. You should see a **Warning** level entry with a **Source** matching the apps namespace. Double-click the entry, or right-click and select **Event Properties** to view the details.



After seeing logs in the **Event Log**, you should stop the service. It's designed to log a random joke once per minute. This is intentional behavior but is *not* practical for production services.

Stop the Windows Service

To stop the Windows Service, use the `sc.exe stop` command:

```
PowerShell  
  
sc.exe stop ".NET Joke Service"
```

You'll see output similar to the following:

```
PowerShell  
  
SERVICE_NAME: .NET Joke Service  
TYPE : 10 WIN32_OWN_PROCESS  
STATE : 3 STOP_PENDING  
        (STOPPABLE, NOT_PAUSABLE, ACCEPTS_SHUTDOWN)  
WIN32_EXIT_CODE : 0 (0x0)  
SERVICE_EXIT_CODE : 0 (0x0)  
CHECKPOINT : 0x0  
WAIT_HINT : 0x0
```

The service **Status** will transition from `STOP_PENDING` to **Stopped**.

Delete the Windows Service

To delete the Windows Service, use the native Windows Service Control Manager's (`sc.exe`) `delete` command. Run PowerShell as an Administrator.

Important

If the service is not in the **Stopped** state, it will not be immediately deleted. Ensure that the service is stopped before issuing the delete command.

PowerShell

```
sc.exe delete ".NET Joke Service"
```

You'll see an output message:

PowerShell

```
[SC] DeleteService SUCCESS
```

For more information, see [sc.exe delete](#).

See also

- [Create a Windows Service installer](#)
- [Worker Services in .NET](#)
- [Create a Queue Service](#)
- [Use scoped services within a `BackgroundService`](#)
- [Implement the `IHostedService` interface](#)

Next

[Create a Windows Service installer](#)

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

.NET is an open source project.
Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

Create a Windows Service installer

Article • 12/13/2023

When you create a .NET Windows Service (not to be mistaken with a .NET Framework Windows Service), you may want to create an installer for your service. Without an installer, users would have to know how to install and configure your service. An installer bundles your app's executables and exposes a customizable installation user experience. This tutorial is a continuation of the [Create a Windows Service](#) tutorial. It shows how to create an installer for your .NET Windows Service.

In this tutorial, you'll learn how to:

- ✓ Install the Visual Studio Installer Projects extension.
- ✓ Create a setup project.
- ✓ Update an existing .NET Worker project to support installation.
- ✓ Automate the installation and uninstallation with the Windows Service Control Manager.

Prerequisites

- You're expected to have completed the [Create a Windows Service](#) tutorial, or be prepared to clone the sample repo.
- The [.NET 8.0 SDK or later](#)
- A Windows OS
- A .NET integrated development environment (IDE)
 - Feel free to use [Visual Studio](#)
- An existing .NET Windows Service

Install tooling dependencies

Wix Toolset

Start by installing the Wix Toolset. The Wix Toolset is a set of tools that build Windows installation packages from XML source code.

.NET CLI

```
dotnet tool install --global wix
```

Next, install the [HeatWave for VS2022 extension](#). After installing, restart Visual Studio and you'll see new project templates available.

Get existing project

This tutorial is based on the app created as part of the [Create a Windows Service using BackgroundService](#) tutorial. You can either clone the sample repo or use the app you built in the previous tutorial.

Tip

All of the "Workers in .NET" example source code is available in the [Samples Browser](#) for download. For more information, see [Browse code samples: Workers in .NET](#).

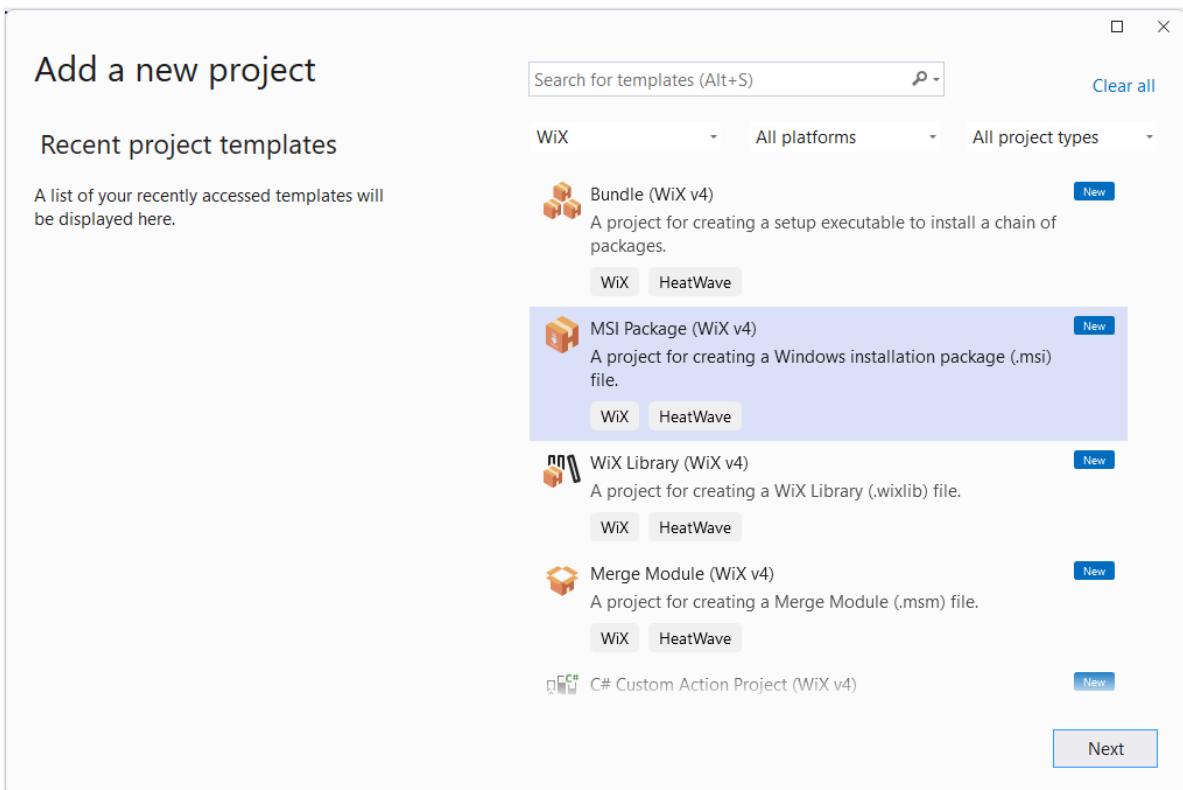
Wix Toolset

Open the solution in Visual Studio, and select `F5` to ensure that the app builds and runs as expected. Press `ctrl + C` to stop the app.

Add new setup project

Wix Toolset

To add a new Wix setup project, right-click on the solution in the [Solution Explorer](#) and select **Add > New Project**:



Select **MSI Package (WiX v4)** from the available templates, then select **Next**. Provide the desired **Name** and **Location**, then select **Create**.

Configure installer project

Wix Toolset

To configure the setup project, you first must add a reference to the `App.WindowsService` project. Right-click the setup project in the **Solution Explorer**, and then select **Add > Project Reference**.

The template includes example component and localization files. Delete these, files leaving only the `Package.wxs` file. Your project should now include a `ProjectReference` element, similar to the following:

XML

```
<Project Sdk="WixToolset.Sdk/4.0.0">
  <ItemGroup>
    <ProjectReference Include=".\\App.WindowsService.csproj" />
  </ItemGroup>
</Project>
```

After the project reference has been added, configure the `Package.wxs` file. Open the file in the editor, and then replace the contents with the following:

XML

```
<?xml version="1.0" encoding="UTF-8"?>

<!-- Define the variables in "$(var.*)" expressions -->
<?define Name = ".NET Joke Service" ?>
<?define Manufacturer = "Microsoft" ?>
<?define Version = "1.0.0.0" ?>
<?define UpgradeCode = "9ED3FF33-8718-444E-B44B-69A2344B7E98" ?>

<Wix xmlns="http://wixtoolset.org/schemas/v4/wxs">
    <Package Name="$(Name)"
        Manufacturer="$(Manufacturer)"
        Version="$(Version)"
        UpgradeCode="$(var.UpgradeCode)"
        Compressed="true">

        <!-- Allow upgrades and prevent downgrades -->
        <MajorUpgrade DowngradeErrorMessage="A later version of
[ProductName] is already installed. Setup will now exit." />

        <!-- Define the directory structure -->
        <Directory Id="TARGETDIR" Name="SourceDir">
            <Directory Id="ProgramFilesFolder">

                <!-- Create a folder inside program files -->
                <Directory Id="ROOTDIRECTORY"
Name="$(var.Manufacturer)">

                    <!-- Create a folder within the parent folder given
the name -->
                    <Directory Id="INSTALLFOLDER" Name="$(Name)" />
                </Directory>
            </Directory>
        </Directory>

        <!-- The files inside this DirectoryRef are linked to
            the App.WindowsService directory via INSTALLFOLDER -->
        <DirectoryRef Id="INSTALLFOLDER">

            <!-- Create a single component which is the
App.WindowsService.exe file -->
            <Component Id="ServiceExecutable" Bitness="always64">

                <!-- Copies the App.WindowsService.exe file using the
                    project reference preprocessor variables -->
                <File Id="App.WindowsService.exe"
Source="$(var.App.WindowsService.TargetDir)publish\App.WindowsService.ex
e"
                    KeyPath="true" />

                <!-- Remove all files from the INSTALLFOLDER on
uninstall -->
            
```

```

<RemoveFile Id="ALLFILES" Name="*.*" On="both" />


<ServiceInstall Id="ServiceInstaller"
    Type="ownProcess"
    Name="App.WindowsService"
    DisplayName="$(Name)"
    Description="A joke service that
periodically logs nerdy humor."
    Start="auto"
    ErrorControl="normal" />


<ServiceControl Id="StartService"
    Start="install"
    Stop="both"
    Remove="uninstall"
    Name="App.WindowsService"
    Wait="true" />
</Component>
</DirectoryRef>


<Feature Id="Service" Title="App.WindowsService Setup"
Level="1">
    <ComponentRef Id="ServiceExecutable" />
</Feature>

</Package>
</Wix>

```

When you build the project, the output is an MSI file that can be used to install and uninstall the service.

Test installation

Wix Toolset

To test the installer, publish the *App.WindowsService* project. Right-click the project in the **Solution Explorer**, and then select **Publish**. Once published with the profile you created in the previous tutorial, the executable will be in the publish directory. Next, you **Build** the setup project and run the installer.

You need to run the installation as an administrator. To do this, right-click the MSI file, and then select **Run as administrator**.

Once the service is installed, you can open **Services** to see the service running. To uninstall the service, use the **Windows Add or Remove Programs** feature to call the installer.

See also

- [Worker Services in .NET](#)
- [Create a Queue Service](#)
- [Use scoped services within a BackgroundService](#)
- [Implement the IHostedService interface](#)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

.NET is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

Implement the `IHostedService` interface

Article • 12/13/2023

When you need finite control beyond the provided [BackgroundService](#), you can implement your own [IHostedService](#). The [IHostedService](#) interface is the basis for all long running services in .NET. Custom implementations are registered with the [AddHostedService<THostedService>\(IServiceCollection\)](#) extension method.

In this tutorial, you learn how to:

- ✓ Implement the [IHostedService](#), and [IAsyncDisposable](#) interfaces.
- ✓ Create a timer-based service.
- ✓ Register the custom implementation with dependency injection and logging.

Tip

All of the "Workers in .NET" example source code is available in the [Samples Browser](#) for download. For more information, see [Browse code samples: Workers in .NET](#).

Prerequisites

- The [.NET 8.0 SDK or later](#)
- A .NET integrated development environment (IDE)
 - Feel free to use [Visual Studio](#)

Create a new project

To create a new Worker Service project with Visual Studio, you'd select **File > New > Project....** From the **Create a new project** dialog search for "Worker Service", and select Worker Service template. If you'd rather use the .NET CLI, open your favorite terminal in a working directory. Run the `dotnet new` command, and replace the `<Project.Name>` with your desired project name.

.NET CLI

```
dotnet new worker --name <Project.Name>
```

For more information on the .NET CLI new worker service project command, see [dotnet new worker](#).

💡 Tip

If you're using Visual Studio Code, you can run .NET CLI commands from the integrated terminal. For more information, see [Visual Studio Code: Integrated Terminal](#).

Create timer service

The timer-based background service makes use of the [System.Threading.Timer](#) class.

The timer triggers the `Dowork` method. The timer is disabled on `IHostLifetime.StopAsync(CancellationToken)` and disposed when the service container is disposed on `IAsyncDisposable.DisposeAsync()`:

Replace the contents of the `Worker` from the template with the following C# code, and rename the file to `TimerService.cs`:

C#

```
namespace App.TIMERHOSTEDSERVICE;

public sealed class TimerService(ILogger<TimerService> logger) : IHostedService, IAsyncDisposable
{
    private readonly Task _completedTask = Task.CompletedTask;
    private int _executionCount = 0;
    private Timer? _timer;

    public Task StartAsync(CancellationToken stoppingToken)
    {
        logger.LogInformation("{Service} is running.", nameof(TIMERHOSTEDSERVICE));
        _timer = new Timer(Dowork, null, TimeSpan.Zero,
TimeSpan.FromSeconds(5));

        return _completedTask;
    }

    private void Dowork(object? state)
    {
        int count = Interlocked.Increment(ref _executionCount);

        logger.LogInformation(
            "{Service} is working, execution count: {Count:#,0}",
            nameof(TIMERHOSTEDSERVICE),
```

```

        count);
    }

    public Task StopAsync(CancellationToken stoppingToken)
    {
        logger.LogInformation(
            "{Service} is stopping.", nameof(TimerHostedService));

        _timer?.Change(Timeout.Infinite, 0);

        return _completedTask;
    }

    public async ValueTask DisposeAsync()
    {
        if (_timer is IAsyncDisposable timer)
        {
            await timer.DisposeAsync();
        }

        _timer = null;
    }
}

```

ⓘ Important

The `Worker` was a subclass of `BackgroundService`. Now, the `TimerService` implements both the `IHostedService`, and `IAsyncDisposable` interfaces.

The `TimerService` is `sealed`, and cascades the `DisposeAsync` call from its `_timer` instance. For more information on the "cascading dispose pattern", see [Implement a `DisposeAsync` method](#).

When `StartAsync` is called, the timer is instantiated, thus starting the timer.

💡 Tip

The `Timer` doesn't wait for previous executions of `DoWork` to finish, so the approach shown might not be suitable for every scenario. `Interlocked.Increment` is used to increment the execution counter as an atomic operation, which ensures that multiple threads don't update `_executionCount` concurrently.

Replace the existing `Program` contents with the following C# code:

C#

```
using App.TimerHostedService;

HostApplicationBuilder builder = Host.CreateApplicationBuilder(args);
builder.Services.AddHostedService<TimerService>();

IHost host = builder.Build();
host.Run();
```

The service is registered in *(Program.cs)* with the `AddHostedService` extension method. This is the same extension method you use when registering [BackgroundService](#) subclasses, as they both implement the [IHostedService](#) interface.

For more information on registering services, see [Dependency injection in .NET](#).

Verify service functionality

To run the application from Visual Studio, select `F5` or select the **Debug > Start Debugging** menu option. If you're using the .NET CLI, run the `dotnet run` command from the working directory:

.NET CLI

```
dotnet run
```

For more information on the .NET CLI run command, see [dotnet run](#).

Let the application run for a bit to generate several execution count increments. You will see output similar to the following:

Output

```
info: App.TimerHostedService.TimerService[0]
      TimerHostedService is running.
info: Microsoft.Hosting.Lifetime[0]
      Application started. Press Ctrl+C to shut down.
info: Microsoft.Hosting.Lifetime[0]
      Hosting environment: Development
info: Microsoft.Hosting.Lifetime[0]
      Content root path: .\timer-service
info: App.TimerHostedService.TimerService[0]
      TimerHostedService is working, execution count: 1
info: App.TimerHostedService.TimerService[0]
      TimerHostedService is working, execution count: 2
info: App.TimerHostedService.TimerService[0]
      TimerHostedService is working, execution count: 3
info: App.TimerHostedService.TimerService[0]
```

```
TimerHostedService is working, execution count: 4
info: Microsoft.Hosting.Lifetime[0]
      Application is shutting down...
info: App.TimerHostedService.TimerService[0]
      TimerHostedService is stopping.
```

If running the application from within Visual Studio, select **Debug > Stop Debugging**.... Alternatively, select **Ctrl + C** from the console window to signal cancellation.

See also

There are several related tutorials to consider:

- [Worker Services in .NET](#)
- [Create a Queue Service](#)
- [Use scoped services within a BackgroundService](#)
- [Create a Windows Service using BackgroundService](#)

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

.NET is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

Deploy a Worker Service to Azure

Article • 12/13/2023

In this article, you'll learn how to deploy a .NET Worker Service to Azure. With your Worker running as an [Azure Container Instance \(ACI\)](#) from the [Azure Container Registry \(ACR\)](#), it can act as a microservice in the cloud. There are many use cases for long-running services, and the Worker Service exists for this reason.

In this tutorial, you learn how to:

- ✓ Create a worker service.
- ✓ Create container registry resource.
- ✓ Push an image to container registry.
- ✓ Deploy as container instance.
- ✓ Verify worker service functionality.

💡 Tip

All of the "Workers in .NET" example source code is available in the [Samples Browser](#) for download. For more information, see [Browse code samples: Workers in .NET](#).

Prerequisites

- The [.NET 5.0 SDK or later](#).
- Docker Desktop ([Windows](#) or [Mac](#)).
- An Azure account with an active subscription. [Create an account for free](#).
- Depending on your developer environment of choice:
 - [Visual Studio or Visual Studio Code](#).
 - [.NET CLI](#)
 - [Azure CLI](#).

Create a new project

To create a new Worker Service project with Visual Studio, select **File > New > Project...**. From the **Create a new project** dialog search for "Worker Service", and select Worker Service template. Enter the desired project name, select an appropriate location, and select **Next**. On the **Additional information** page, for the **Target Framework** select [.NET](#)

5.0, and check the **Enable Docker** option to enable docker support. Select the desired Docker OS.

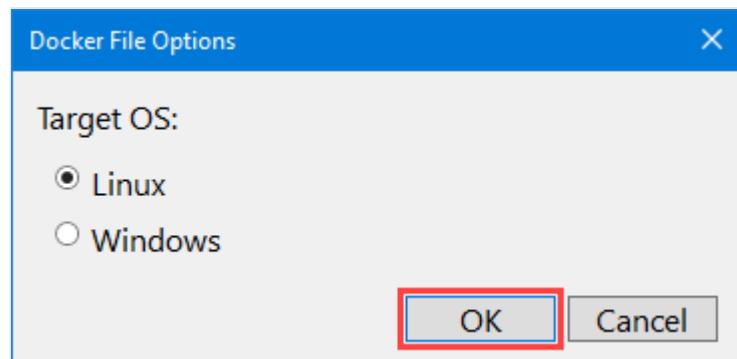
Build the application to ensure it restores the dependent packages, and compiles without error.

To build the application from Visual Studio, select **F6** or select the **Build > Build Solution** menu option.

Add Docker support

If you correctly selected the **Enable Docker** checkbox when creating a new Worker project, skip to the [Build the Docker image](#) step.

If you didn't select this option, no worries—you can still add it now. In Visual Studio, right-click on the *project node* in the **Solution Explorer**, and select **Add > Docker Support**. You'll be prompted to select a **Target OS**; select **OK** with the default OS selection.



Docker support requires a *Dockerfile*. This file is a set of comprehensive instructions, for building your .NET Worker Service as a Docker image. The *Dockerfile* is a file *without* a file extension. The following code is an example *Dockerfile*, and should exist at the root directory of the project file.

```
Dockerfile

FROM mcr.microsoft.com/dotnet/runtime:8.0 AS base
WORKDIR /app

# Creates a non-root user with an explicit UID and adds permission to access
# the /app folder
# For more info, please refer to https://aka.ms/vscode-docker-dotnet-
# configure-containers
RUN adduser -u 5678 --disabled-password --gecos "" appuser && chown -R
appuser /app
USER appuser
```

```
FROM mcr.microsoft.com/dotnet/sdk:8.0 AS build
WORKDIR /src
COPY ["App.CloudService.csproj", "./"]
RUN dotnet restore "App.CloudService.csproj"
COPY . .
WORKDIR "/src/."
RUN dotnet build "App.CloudService.csproj" -c Release -o /app/build

FROM build AS publish
RUN dotnet publish "App.CloudService.csproj" -c Release -o /app/publish

FROM base AS final
WORKDIR /app
COPY --from=publish /app/publish .
ENTRYPOINT ["dotnet", "App.CloudService.dll"]
```

Build the Docker image

To build the Docker image, the Docker Engine must be running.

ⓘ Important

When using Docker Desktop and Visual Studio, to avoid errors related to *volume sharing* — ensure that it is enabled.

1. On the **Settings** screen in Docker Desktop, select **Shared Drives**.
2. Select the drive(s) containing your project files.

For more information, see [Troubleshoot Visual Studio development with Docker](#).

Right-click on the *Dockerfile* in the **Solution Explorer**, and select **Build Docker Image**. The **Output** window displays, reporting the `docker build` command progress.

As the `docker build` command runs, it processes each line in the *Dockerfile* as an instruction step. This command builds the image and creates a local repository named `appcloudservice` that points to the image.

💡 Tip

The generated *Dockerfile* differs between development environments. For example, if you **Add Docker support** from Visual Studio you may experience issues if you attempt to **Build the Docker image** from Visual Studio Code—as the *Dockerfile* steps vary. It is best to choose a single *development environment* and use it throughout this tutorial.

Create container registry

An Azure Container Registry (ACR) resource allows you to build, store, and manage container images and artifacts in a private registry. To create a container registry, you need to [create a new resource](#) in the Azure portal.

1. Select the **Subscription**, and corresponding **Resource group** (or create a new one).
2. Enter a **Registry name**.
3. Select a **Location**.
4. Select an appropriate **SKU**, for example **Basic**.
5. Select **Review + create**.
6. After seeing **Validation passed**, select **Create**.

ⓘ Important

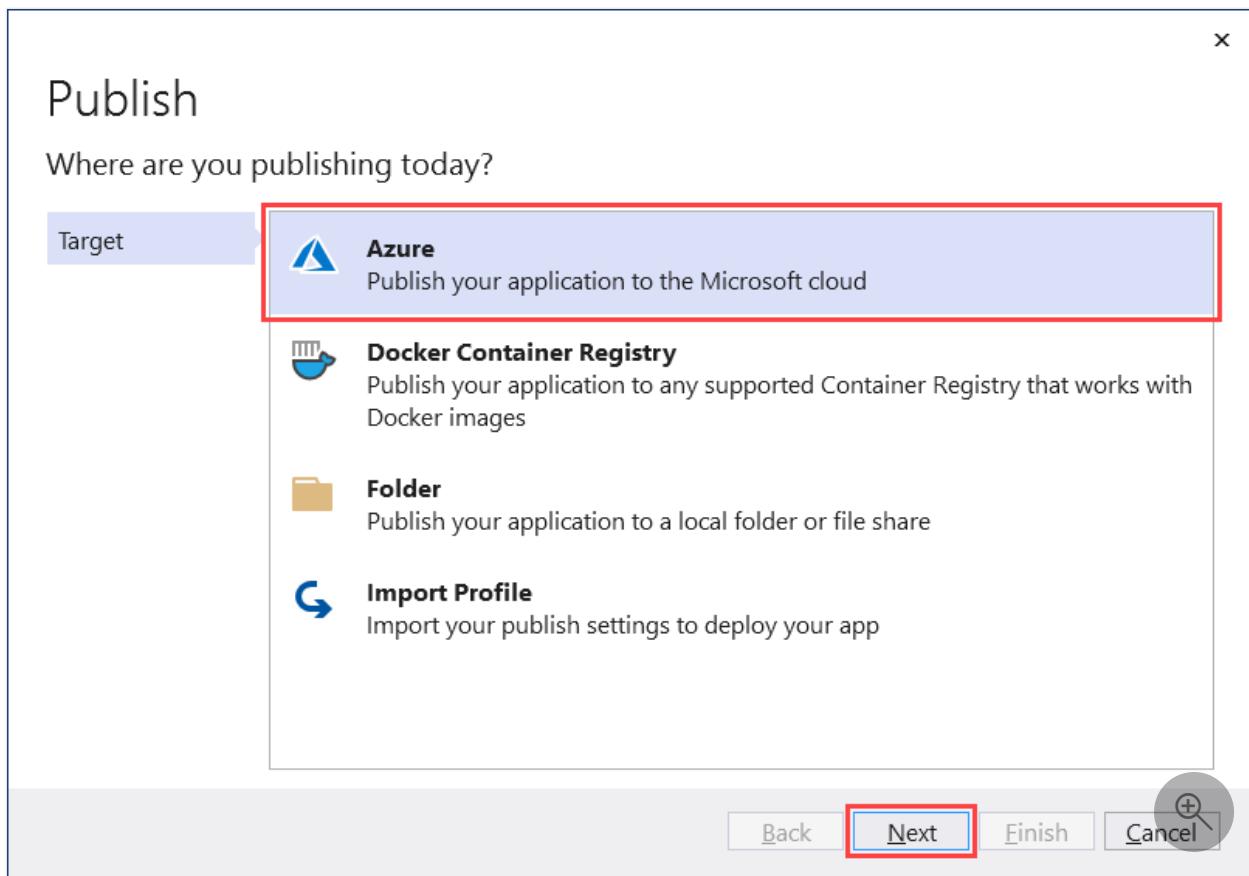
In order to use this container registry when creating a container instance, you must enable **Admin user**. Select **Access keys**, and enable **Admin user**.

For more information, see [Quickstart: Create an Azure container registry](#).

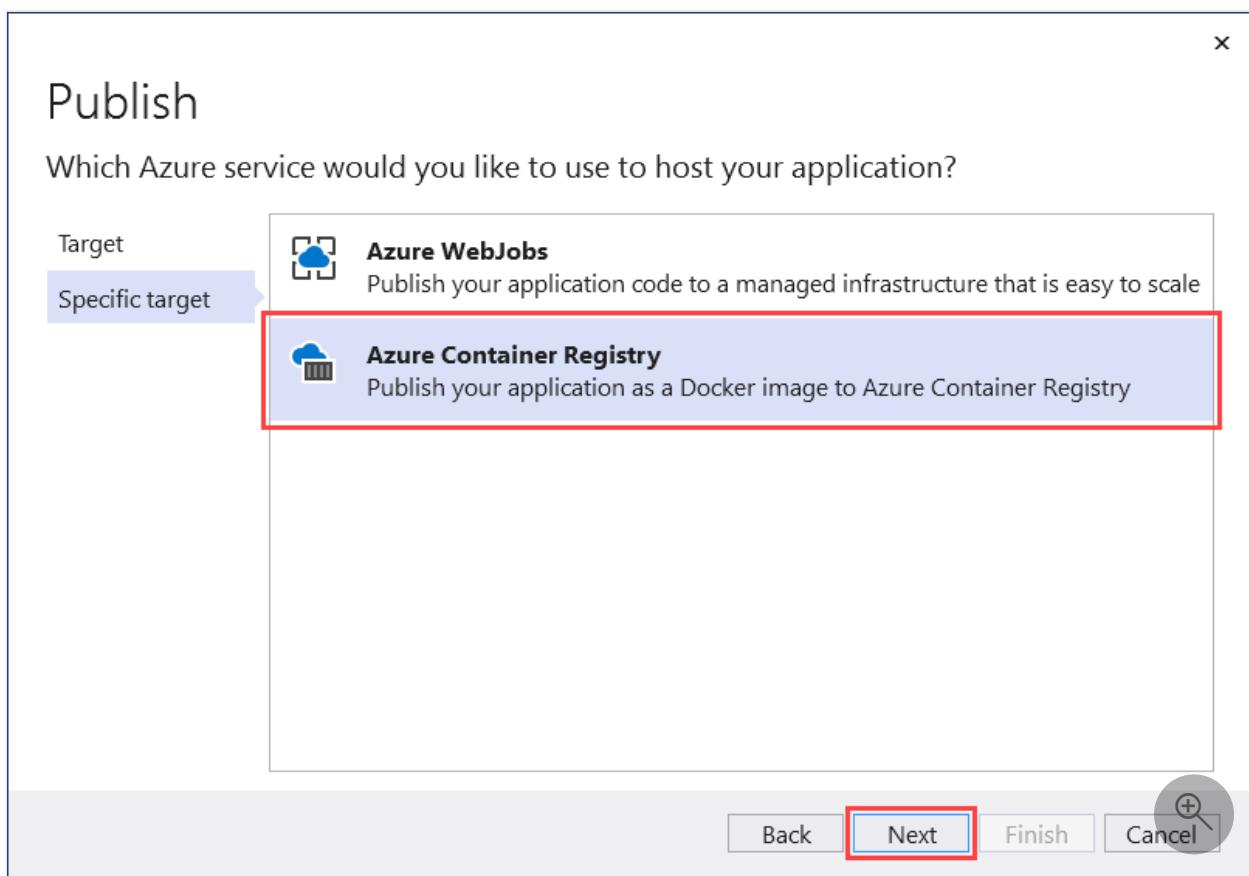
Push image to container registry

With the .NET Docker image built, and the container registry resource created, you can now push the image to container registry.

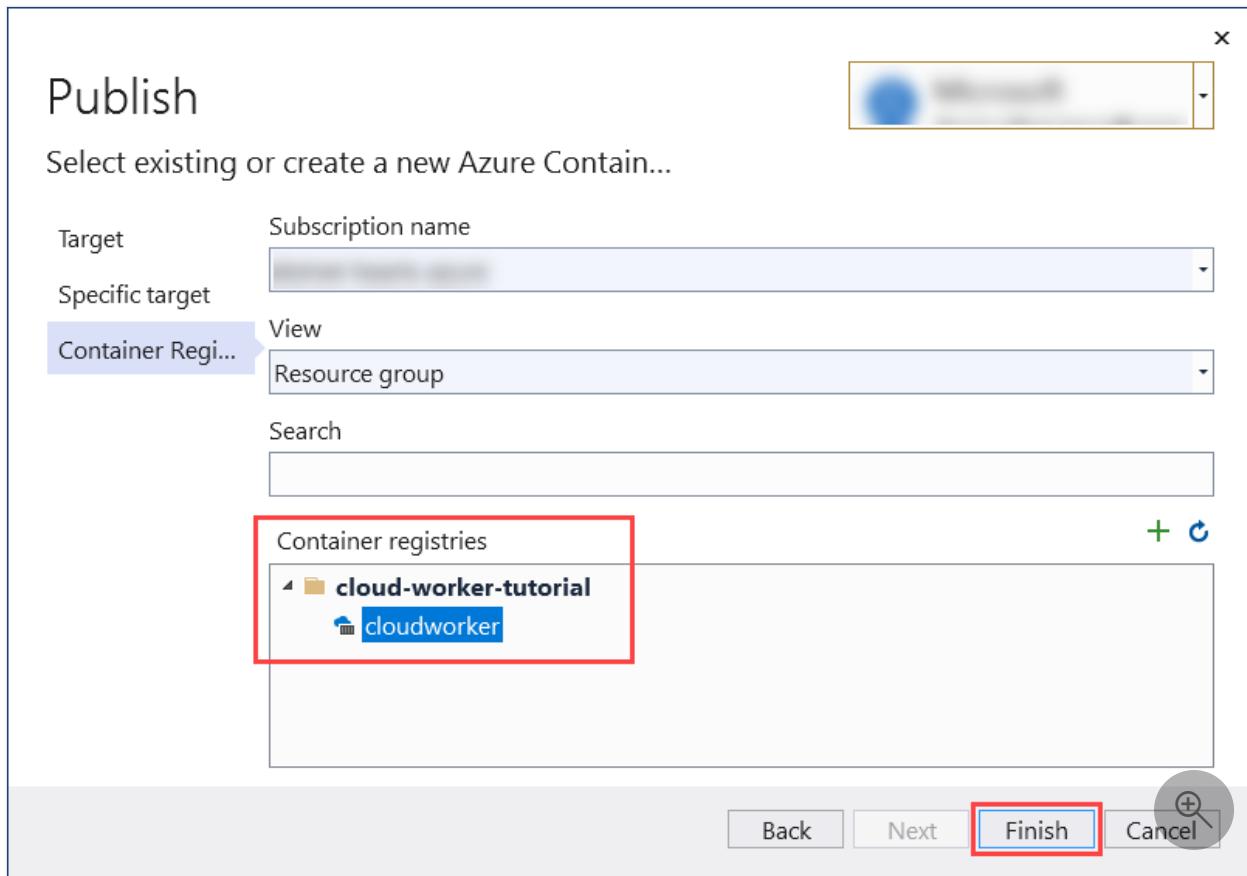
Right-click on the project in the **Solution Explorer**, and select **Publish**. The **Publish** dialog displays. For the **Target**, select **Azure** and then **Next**.



For the **Specific Target**, select **Azure Container Registry** and then **Next**.



Next, for the **Container Registry**, select the **Subscription name** that you used to create the ACR resource. From the **Container registries** selection area, select the container registry that you created, and then select **Finish**.



This creates a publish profile, which can be used to publish the image to container registry. Select the **Publish** button to push the image to the container registry, the **Output** window reports the publish progress—and when it completes successfully, you'll see a "Successfully published" message.

To verify that the image was successfully pushed to the container registry, navigate to the Azure portal. Open the container registry resource, under **Services**, select **Repositories**. You should see the image.

Deploy as container instance

To create a container instance, you also need to [create a new resource](#) in the Azure portal.

1. Select the same **Subscription**, and corresponding **Resource group** from the previous section.
2. Enter a **Container name**—`appcloudservice-container`.
3. Select a **Region** that corresponds to the previous **Location** selection.
4. For **Image source**, select **Azure Container Registry**.
5. Select the **Registry** by the name provided in the previous step.
6. Select the **Image** and **Image tag**.
7. Select **Review + create**.
8. Assuming **Validation passed**, select **Create**.

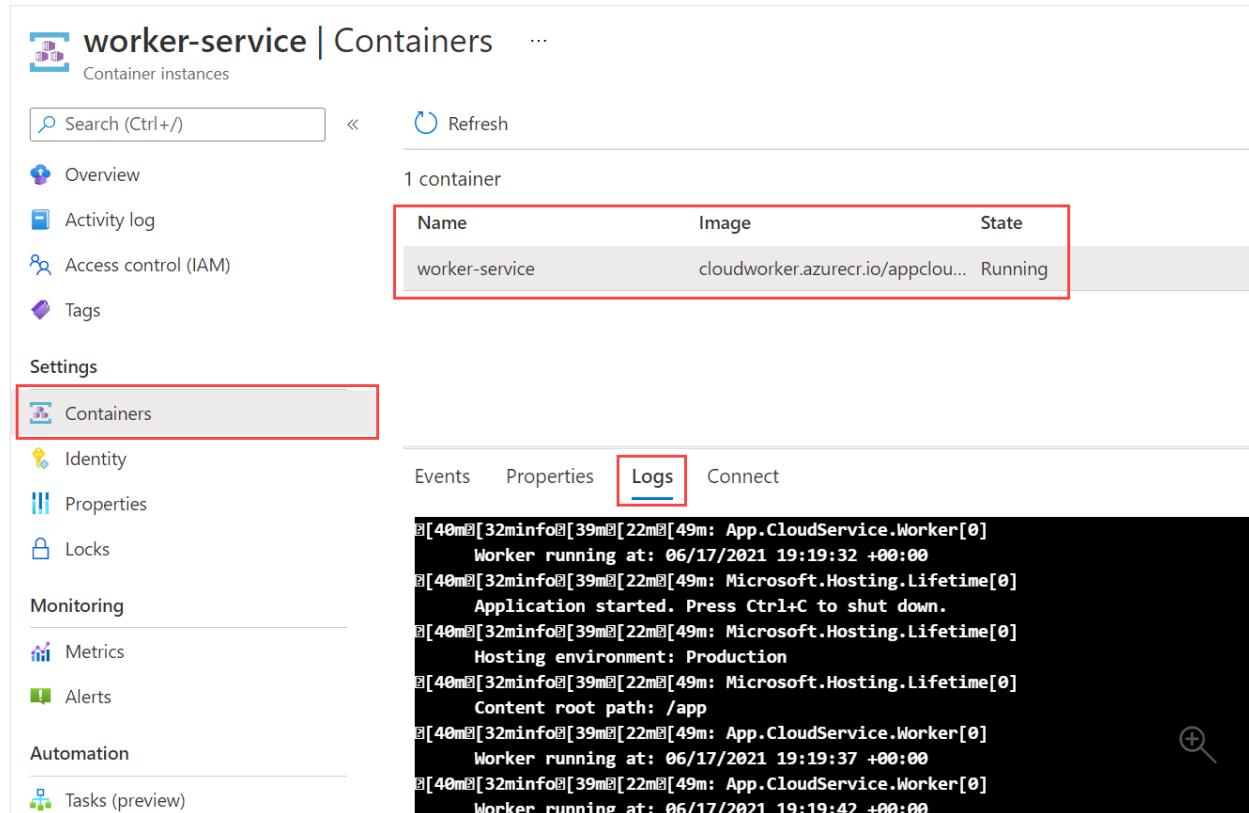
It may take a moment for the resources to be created, once created select the **Go to resource** button.

For more information, see [Quickstart: Create an Azure container instance](#).

Verify service functionality

Immediately after the container instance is created, it starts running.

To verify your worker service is functioning correctly, navigate to the Azure portal in the container instance resource, select the **Containers** option.



The screenshot shows the Azure portal interface for a container instance named "worker-service". The left sidebar has sections for Overview, Activity log, Access control (IAM), Tags, Settings (with Containers selected), Identity, Properties, Locks, Monitoring (Metrics and Alerts), Automation, and Tasks (preview). The main content area shows "1 container" and a table with columns Name, Image, and State. A single row is shown: Name is "worker-service", Image is "cloudworker.azurecr.io/appclou...", and State is "Running". Below the table is a "Logs" section with tabs for Events, Properties, Logs (selected), and Connect. The logs pane displays several lines of .NET worker service output:

```
[40m][32minfo@[39m@[22m@[49m: App.CloudService.Worker[0]
    Worker running at: 06/17/2021 19:19:32 +00:00
[40m][32minfo@[39m@[22m@[49m: Microsoft.Hosting.Lifetime[0]
    Application started. Press Ctrl+C to shut down.
[40m][32minfo@[39m@[22m@[49m: Microsoft.Hosting.Lifetime[0]
    Hosting environment: Production
[40m][32minfo@[39m@[22m@[49m: Microsoft.Hosting.Lifetime[0]
    Content root path: /app
[40m][32minfo@[39m@[22m@[49m: App.CloudService.Worker[0]
    Worker running at: 06/17/2021 19:19:37 +00:00
[40m][32minfo@[39m@[22m@[49m: App.CloudService.Worker[0]
    Worker running at: 06/17/2021 19:19:42 +00:00
```

You'll see the containers and their current **State**. In this case, it is **Running**. Select **Logs** to see the .NET worker service output.

See also

- [Worker Services in .NET](#)
- [Use scoped services within a BackgroundService](#)
- [Create a Windows Service using BackgroundService](#)
- [Implement the IHostedService interface](#)
- [Tutorial: Containerize a .NET Core app](#)

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

.NET is an open source project.
Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

Caching in .NET

Article • 12/11/2023

In this article, you'll learn about various caching mechanisms. Caching is the act of storing data in an intermediate-layer, making subsequent data retrievals faster.

Conceptually, caching is a performance optimization strategy and design consideration. Caching can significantly improve app performance by making infrequently changing (or expensive to retrieve) data more readily available. This article introduces the two primary types of caching, and provides sample source code for both:

- [Microsoft.Extensions.Caching.Memory](#)
- [Microsoft.Extensions.Caching.Distributed](#)

ⓘ Important

There are two `MemoryCache` classes within .NET, one in the `System.Runtime.Caching` namespace and the other in the `Microsoft.Extensions.Caching` namespace:

- [System.Runtime.Caching.MemoryCache](#)
- [Microsoft.Extensions.Caching.Memory.MemoryCache](#)

While this article focuses on caching, it doesn't include the [System.Runtime.Caching](#) NuGet package. All references to `MemoryCache` are within the `Microsoft.Extensions.Caching` namespace.

All of the `Microsoft.Extensions.*` packages come dependency injection (DI) ready, both the `IMemoryCache` and `IDistributedCache` interfaces can be used as services.

In-memory caching

In this section, you'll learn about the [Microsoft.Extensions.Caching.Memory](#) package. The current implementation of the `IMemoryCache` is a wrapper around the `ConcurrentDictionary<TKey,TValue>`, exposing a feature-rich API. Entries within the cache are represented by the `ICacheEntry`, and can be any `object`. The in-memory cache solution is great for apps that run on a single server, where all the cached data rents memory in the app's process.

💡 Tip

For multi-server caching scenarios, consider the [Distributed caching](#) approach as an alternative to in-memory caching.

In-memory caching API

The consumer of the cache has control over both sliding and absolute expirations:

- [ICacheEntry.AbsoluteExpiration](#)
- [ICacheEntry.AbsoluteExpirationRelativeToNow](#)
- [ICacheEntry.SlidingExpiration](#)

Setting an expiration will cause entries in the cache to be *evicted* if they're not accessed within the expiration time allotment. Consumers have additional options for controlling cache entries, through the [MemoryCacheEntryOptions](#). Each [ICacheEntry](#) is paired with [MemoryCacheEntryOptions](#) which exposes expiration eviction functionality with [IChangeToken](#), priority settings with [CacheItemPriority](#), and controlling the [ICacheEntry.Size](#). Consider the following extension methods:

- [MemoryCacheEntryExtensions.AddExpirationToken](#)
- [MemoryCacheEntryExtensions.RegisterPostEvictionCallback](#)
- [MemoryCacheEntryExtensions.SetSize](#)
- [MemoryCacheEntryExtensions.SetPriority](#)

In-memory cache example

To use the default [IMemoryCache](#) implementation, call the [AddMemoryCache](#) extension method to register all the required services with DI. In the following code sample, the generic host is used to expose DI functionality:

C#

```
using Microsoft.Extensions.Caching.Memory;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;

HostApplicationBuilder builder = Host.CreateApplicationBuilder(args);
builder.Services.AddMemoryCache();
using IHost host = builder.Build();
```

Depending on your .NET workload, you may access the [IMemoryCache](#) differently; such as constructor injection. In this sample, you use the [IServiceProvider](#) instance on the [host](#) and call generic [GetRequiredService<T>\(IServiceProvider\)](#) extension method:

```
C#
```

```
IMemoryCache cache =  
    host.Services.GetRequiredService<IMemoryCache>();
```

With in-memory caching services registered, and resolved through DI — you're ready to start caching. This sample iterates through the letters in the English alphabet 'A' through 'Z'. The `record AlphabetLetter` type holds the reference to the letter, and generates a message.

```
C#
```

```
file record AlphabetLetter(char Letter)  
{  
    internal string Message =>  
        $"The '{Letter}' character is the {Letter - 64} letter in the  
        English alphabet.";  
}
```

The sample includes a helper function that iterates through the alphabet letters:

```
C#
```

```
static async ValueTask IterateAlphabetAsync(  
    Func<char, Task> asyncFunc)  
{  
    for (char letter = 'A'; letter <= 'Z'; ++letter)  
    {  
        await asyncFunc(letter);  
    }  
  
    Console.WriteLine();  
}
```

In the preceding C# code:

- The `Func<char, Task> asyncFunc` is awaited on each iteration, passing the current `letter`.
- After all letters have been processed, a blank line is written to the console.

To add items to the cache call one of the `Create`, or `Set` APIs:

```
C#
```

```
var addLettersToCacheTask = IterateAlphabetAsync(letter =>  
{  
    MemoryCacheEntryOptions options = new()
```

```

{
    AbsoluteExpirationRelativeToNow =
        TimeSpan.FromMilliseconds(MillisecondsAbsoluteExpiration)
};

_ = options.RegisterPostEvictionCallback(OnPostEviction);

AlphabetLetter alphabetLetter =
    cache.Set(
        letter, new AlphabetLetter(letter), options);

Console.WriteLine($"{alphabetLetter.Letter} was cached.");

return Task.Delay(
    TimeSpan.FromMilliseconds(MillisecondsDelayAfterAdd));
});

await addLettersToCacheTask;

```

In the preceding C# code:

- The variable `addLettersToCacheTask` delegates to `IterateAlphabetAsync` and is awaited.
- The `Func<char, Task> asyncFunc` is argued with a lambda.
- The `MemoryCacheEntryOptions` is instantiated with an absolute expiration relative to now.
- A post eviction callback is registered.
- An `AlphabetLetter` object is instantiated, and passed into `Set` along with `letter` and `options`.
- The letter is written to the console as being cached.
- Finally, a `Task.Delay` is returned.

For each letter in the alphabet, a cache entry is written with an expiration, and post eviction callback.

The post eviction callback writes the details of the value that was evicted to the console:

C#

```

static void OnPostEviction(
    object key, object? letter, EvictionReason reason, object? state)
{
    if (letter is AlphabetLetter alphabetLetter)
    {
        Console.WriteLine($"{alphabetLetter.Letter} was evicted for
{reason}.");
    }
}

```

Now that the cache is populated, another call to `IterateAlphabetAsync` is awaited, but this time you'll call `IMemoryCache.TryGetValue`:

```
C#  
  
var readLettersFromCacheTask = IterateAlphabetAsync(letter =>  
{  
    if (cache.TryGetValue(letter, out object? value) &&  
        value is AlphabetLetter alphabetLetter)  
    {  
        Console.WriteLine($"{letter} is still in cache.  
{alphabetLetter.Message}");  
    }  
  
    return Task.CompletedTask;  
});  
await readLettersFromCacheTask;
```

If the `cache` contains the `letter` key, and the `value` is an instance of an `AlphabetLetter` it's written to the console. When the `letter` key is not in the cache, it was evicted and its post eviction callback was invoked.

Additional extension methods

The `IMemoryCache` comes with many convenience-based extension methods, including an asynchronous `GetOrCreateAsync`:

- [CacheExtensions.Get](#)
- [CacheExtensions.GetOrCreate](#)
- [CacheExtensions.GetOrCreateAsync](#)
- [CacheExtensions.Set](#)
- [CacheExtensions.TryGetValue](#)

Put it all together

The entire sample app source code is a top-level program and requires two NuGet packages:

- [Microsoft.Extensions.Caching.Memory](#)
- [Microsoft.Extensions.Hosting](#)

```
C#  
  
using Microsoft.Extensions.Caching.Memory;  
using Microsoft.Extensions.DependencyInjection;
```

```
using Microsoft.Extensions.Hosting;

HostApplicationBuilder builder = Host.CreateApplicationBuilder(args);
builder.Services.AddMemoryCache();
using IHost host = builder.Build();

IMemoryCache cache =
    host.Services.GetRequiredService<IMemoryCache>();

const int MillisecondsDelayAfterAdd = 50;
const int MillisecondsAbsoluteExpiration = 750;

static void OnPostEviction(
    object key, object? letter, EvictionReason reason, object? state)
{
    if (letter is AlphabetLetter alphabetLetter)
    {
        Console.WriteLine($"'{alphabetLetter.Letter}' was evicted for
{reason}.");
    }
};

static async ValueTask IterateAlphabetAsync(
    Func<char, Task> asyncFunc)
{
    for (char letter = 'A'; letter <= 'Z'; ++letter)
    {
        await asyncFunc(letter);
    }

    Console.WriteLine();
}

var addLettersToCacheTask = IterateAlphabetAsync(letter =>
{
    MemoryCacheEntryOptions options = new()
    {
        AbsoluteExpirationRelativeToNow =
            TimeSpan.FromMilliseconds(MillisecondsAbsoluteExpiration)
    };

    _ = options.RegisterPostEvictionCallback(OnPostEviction);

    AlphabetLetter alphabetLetter =
        cache.Set(
            letter, new AlphabetLetter(letter), options);

    Console.WriteLine($"'{alphabetLetter.Letter}' was cached.");
}

    return Task.Delay(
        TimeSpan.FromMilliseconds(MillisecondsDelayAfterAdd));
});

await addLettersToCacheTask;

var readLettersFromCacheTask = IterateAlphabetAsync(letter =>
```

```
{  
    if (cache.TryGetValue(letter, out object? value) &&  
        value is AlphabetLetter alphabetLetter)  
    {  
        Console.WriteLine($"{letter} is still in cache.  
{alphabetLetter.Message}");  
    }  
  
    return Task.CompletedTask;  
});  
await readLettersFromCacheTask;  
  
await host.RunAsync();  
  
file record AlphabetLetter(char Letter)  
{  
    internal string Message =>  
        $"The '{Letter}' character is the {Letter - 64} letter in the  
English alphabet.";  
}
```

Feel free to adjust the `MillisecondsDelayAfterAdd` and `MillisecondsAbsoluteExpiration` values to observe the changes in behavior to the expiration and eviction of cached entries. The following is sample output from running this code, due to the non-deterministic nature of .NET events — there is no guarantee that your output will be identical.

Console

```
A was cached.  
B was cached.  
C was cached.  
D was cached.  
E was cached.  
F was cached.  
G was cached.  
H was cached.  
I was cached.  
J was cached.  
K was cached.  
L was cached.  
M was cached.  
N was cached.  
O was cached.  
P was cached.  
Q was cached.  
R was cached.  
S was cached.  
T was cached.  
U was cached.  
V was cached.  
W was cached.
```

```
X was cached.  
Y was cached.  
Z was cached.  
  
A was evicted for Expired.  
C was evicted for Expired.  
B was evicted for Expired.  
E was evicted for Expired.  
D was evicted for Expired.  
F was evicted for Expired.  
H was evicted for Expired.  
K was evicted for Expired.  
L was evicted for Expired.  
J was evicted for Expired.  
G was evicted for Expired.  
M was evicted for Expired.  
N was evicted for Expired.  
I was evicted for Expired.  
P was evicted for Expired.  
R was evicted for Expired.  
O was evicted for Expired.  
Q was evicted for Expired.  
S is still in cache. The 'S' character is the 19 letter in the English  
alphabet.  
T is still in cache. The 'T' character is the 20 letter in the English  
alphabet.  
U is still in cache. The 'U' character is the 21 letter in the English  
alphabet.  
V is still in cache. The 'V' character is the 22 letter in the English  
alphabet.  
W is still in cache. The 'W' character is the 23 letter in the English  
alphabet.  
X is still in cache. The 'X' character is the 24 letter in the English  
alphabet.  
Y is still in cache. The 'Y' character is the 25 letter in the English  
alphabet.  
Z is still in cache. The 'Z' character is the 26 letter in the English  
alphabet.
```

Since the absolute expiration

([MemoryCacheEntryOptions.AbsoluteExpirationRelativeToNow](#)) is set, all the cached items will eventually be evicted.

Worker Service caching

One common strategy for caching data, is updating the cache independently from the consuming data services. The *Worker Service* template is a great example, as the [BackgroundService](#) runs independent (or in the background) from the other application code. When an application starts running that hosts an implementation of the [IHostedService](#), the corresponding implementation (in this case the [BackgroundService](#)

or "worker") start running in the same process. These hosted services are registered with DI as singletons, through the `AddHostedService<THostedService>(IServiceCollection)` extension method. Other services can be registered with DI with any [service lifetime](#).

ⓘ Important

The service lifetime's are very important to understand. When you call `AddMemoryCache` to register all of the in-memory caching services, the services are registered as singletons.

Photo service scenario

Imagine you're developing a photo service that relies on third-party API accessible via HTTP. This photo data doesn't change very often, but there is a lot of it. Each photo is represented by a simple `record`:

C#

```
namespace CachingExamples.Memory;

public readonly record struct Photo(
    int AlbumId,
    int Id,
    string Title,
    string Url,
    string ThumbnailUrl);
```

In the following example, you'll see several services being registered with DI. Each service has a single responsibility.

C#

```
using CachingExamples.Memory;

HostApplicationBuilder builder = Host.CreateApplicationBuilder(args);

builder.Services.AddMemoryCache();
builder.Services.AddHttpClient<CacheWorker>();
builder.Services.AddHostedService<CacheWorker>();
builder.Services.AddScoped<PhotoService>();
builder.Services.AddSingleton(typeof(CacheSignal<>));

using IHost host = builder.Build();

await host.StartAsync();
```

In the preceding C# code:

- The generic host is created with [defaults](#).
- In-memory caching services are registered with [AddMemoryCache](#).
- An `HttpClient` instance is registered for the `CacheWorker` class with [AddHttpClient<TClient>\(IServiceCollection\)](#).
- The `CacheWorker` class is registered with [AddHostedService<THostedService>\(IServiceCollection\)](#).
- The `PhotoService` class is registered with [AddScoped<TService>\(IServiceCollection\)](#).
- The `CacheSignal<T>` class is registered with [AddSingleton](#).
- The `host` is instantiated from the builder, and started asynchronously.

The `PhotoService` is responsible for getting photos that match given criteria (or `filter`):

C#

```
using Microsoft.Extensions.Caching.Memory;

namespace CachingExamples.Memory;

public sealed class PhotoService(
    IMemoryCache cache,
    CacheSignal<Photo> cacheSignal,
    ILogger<PhotoService> logger)
{
    public async IAsyncEnumerable<Photo> GetPhotosAsync(Func<Photo, bool>?
filter = default)
    {
        try
        {
            await cacheSignal.WaitAsync();

            Photo[] photos =
                (await cache.GetOrCreateAsync(
                    "Photos", _ =>
                {
                    logger.LogWarning("This should never happen!");
                })
            )!;

            // If no filter is provided, use a pass-thru.
            filter ??= _ => true;

            foreach (Photo photo in photos)
            {
                if (!default(Photo).Equals(photo) && filter(photo))
                {
                    yield return photo;
                }
            }
        }
    }
}
```

```

        }
    }
    finally
    {
        cacheSignal.Release();
    }
}
}

```

In the preceding C# code:

- The constructor requires an `IMemoryCache`, `CacheSignal<Photo>`, and `ILogger`.
- The `GetPhotosAsync` method:
 - Defines a `Func<Photo, bool> filter` parameter, and returns an `IAsyncEnumerable<Photo>`.
 - Calls and waits for the `_cacheSignal.WaitAsync()` to release, this ensures that the cache is populated before accessing the cache.
 - Calls `_cache.GetOrCreateAsync()`, asynchronously getting all of the photos in the cache.
 - The `factory` argument logs a warning, and returns an empty photo array - this should never happen.
 - Each photo in the cache is iterated, filtered, and materialized with `yield return`.
 - Finally, the cache signal is reset.

Consumers of this service are free to call `GetPhotosAsync` method, and handle photos accordingly. No `HttpClient` is required as the cache contains the photos.

The asynchronous signal is based on an encapsulated `SemaphoreSlim` instance, within a generic-type constrained singleton. The `CacheSignal<T>` relies on an instance of `SemaphoreSlim`:

C#

```

namespace CachingExamples.Memory;

public sealed class CacheSignal<T>
{
    private readonly SemaphoreSlim _semaphore = new(1, 1);

    /// <summary>
    /// Exposes a <see cref="Task"/> that represents the asynchronous wait
    /// operation.
    /// When signaled (consumer calls <see cref="Release"/>), the
    /// <see cref="Task.Status"/> is set as <see
    /// cref="TaskStatus.RanToCompletion"/>.
}

```

```

    /// </summary>
    public Task WaitAsync() => _semaphore.WaitAsync();

    /// <summary>
    /// Exposes the ability to signal the release of the <see
    cref="WaitAsync"/>'s operation.
    /// Callers who were waiting, will be able to continue.
    /// </summary>
    public void Release() => _semaphore.Release();
}

```

In the preceding C# code, the decorator pattern is used to wrap an instance of the `SemaphoreSlim`. Since the `CacheSignal<T>` is registered as a singleton, it can be used across all service lifetimes with any generic type — in this case, the `Photo`. It is responsible for signaling the seeding of the cache.

The `CacheWorker` is a subclass of `BackgroundService`:

C#

```

using System.Net.Http.Json;
using Microsoft.Extensions.Caching.Memory;

namespace CachingExamples.Memory;

public sealed class CacheWorker(
    ILogger<CacheWorker> logger,
    HttpClient httpClient,
    CacheSignal<Photo> cacheSignal,
    IMemoryCache cache) : BackgroundService
{
    private readonly TimeSpan _updateInterval = TimeSpan.FromHours(3);

    private bool _isCacheInitialized = false;

    private const string Url =
"https://jsonplaceholder.typicode.com/photos";

    public override async Task StartAsync(CancellationToken cancellationToken)
    {
        await cacheSignal.WaitAsync();
        await base.StartAsync(cancellationToken);
    }

    protected override async Task ExecuteAsync(CancellationToken stoppingToken)
    {
        while (!stoppingToken.IsCancellationRequested)
        {
            logger.LogInformation("Updating cache.");
        }
    }
}

```

```

    try
    {
        Photo[]? photos =
            await httpClient.GetFromJsonAsync<Photo[]>(
                Url, stoppingToken);

        if (photos is { Length: > 0 })
        {
            cache.Set("Photos", photos);
            logger.LogInformation(
                "Cache updated with {Count:#,#} photos.",
                photos.Length);
        }
        else
        {
            logger.LogWarning(
                "Unable to fetch photos to update cache.");
        }
    }
    finally
    {
        if (!_isCacheInitialized)
        {
            cacheSignal.Release();
            _isCacheInitialized = true;
        }
    }

    try
    {
        logger.LogInformation(
            "Will attempt to update the cache in {Hours} hours from
now.",
            _updateInterval.Hours);

        await Task.Delay(_updateInterval, stoppingToken);
    }
    catch (OperationCanceledException)
    {
        logger.LogWarning("Cancellation acknowledged: shutting
down.");
        break;
    }
}
}
}

```

In the preceding C# code:

- The constructor requires an `ILogger`, `HttpClient`, and `IMemoryCache`.
- The `_updateInterval` is defined for three hours.
- The `ExecuteAsync` method:

- Loops while the app is running.
- Makes an HTTP request to "<https://jsonplaceholder.typicode.com/photos>", and maps the response as an array of `Photo` objects.
- The array of photos is placed in the `IMemoryCache` under the "Photos" key.
- The `_cacheSignal.Release()` is called, releasing any consumers who were waiting for the signal.
- The call to `Task.Delay` is awaited, given the update interval.
- After delaying for three hours, the cache is again updated.

Consumers in the same process could ask the `IMemoryCache` for the photos, but the `CacheWorker` is responsible for updating the cache.

Distributed caching

In some scenarios, a distributed cache is required — such is the case with multiple app servers. A distributed cache supports higher scale-out than the in-memory caching approach. Using a distributed cache offloads the cache memory to an external process, but does require extra network I/O and introduces a bit more latency (even if nominal).

The distributed caching abstractions are part of the [Microsoft.Extensions.Caching.Memory](#) NuGet package, and there is even an `AddDistributedMemoryCache` extension method.

⊗ Caution

The `AddDistributedMemoryCache` should only be used in development and/or testing scenarios, and is **not** a viable production implementation.

Consider any of the available implementations of the `IDistributedCache` from the following packages:

- [Microsoft.Extensions.Caching.SqlServer](#)
- [Microsoft.Extensions.Caching.StackExchangeRedis](#)
- [NCache.Microsoft.Extensions.Caching.OpenSource](#)

Distributed caching API

The distributed caching APIs are a bit more primitive than their in-memory caching API counterparts. The key-value pairs are a bit more basic. In-memory caching keys are based on an `object`, whereas the distributed keys are a `string`. With in-memory

caching, the value can be any strongly-typed generic, whereas values in distributed caching are persisted as `byte[]`. That's not to say that various implementations don't expose strongly-typed generic values but that would be an implementation detail.

Create values

To create values in the distributed cache, call one of the set APIs:

- [IDistributedCache.SetAsync](#)
- [IDistributedCache.Set](#)

Using the `AlphabetLetter` record from the in-memory cache example, you could serialize the object to JSON and then encode the `string` as a `byte[]`:

C#

```
DistributedCacheEntryOptions options = new()
{
    AbsoluteExpirationRelativeToNow =
        TimeSpan.FromMilliseconds(MillisecondsAbsoluteExpiration)
};

AlphabetLetter alphabetLetter = new(letter);
string json = JsonSerializer.Serialize(alphabetLetter);
byte[] bytes = Encoding.UTF8.GetBytes(json);

await cache.SetAsync(letter.ToString(), bytes, options);
```

Much like in-memory caching, cache entries can have options to help fine-tune their existence in the cache — in this case, the [DistributedCacheEntryOptions](#).

Create extension methods

There are several convenience-based extension methods for creating values, that help to avoid encoding `string` representations of objects into a `byte[]`:

- [DistributedCacheExtensions.SetStringAsync](#)
- [DistributedCacheExtensions.SetString](#)

Read values

To read values from the distributed cache, call one of the get APIs:

- [IDistributedCache.GetAsync](#)
- [IDistributedCache.Get](#)

C#

```
AlphabetLetter? alphabetLetter = null;
byte[]? bytes = await cache.GetAsync(letter.ToString());
if (bytes is { Length: > 0 })
{
    string json = Encoding.UTF8.GetString(bytes);
    alphabetLetter = JsonSerializer.Deserialize<AlphabetLetter>(json);
}
```

Once a cache entry is read out of the cache, you can get the UTF8 encoded `string` representation from the `byte[]`

Read extension methods

There are several convenience-based extension methods for reading values, that help to avoid decoding `byte[]` into `string` representations of objects:

- [DistributedCacheExtensions.GetStringAsync](#)
- [DistributedCacheExtensions.GetString](#)

Update values

There is no way to update the values in the distributed cache with a single API call, instead, values can have their sliding expirations reset with one of the refresh APIs:

- [IDistributedCache.RefreshAsync](#)
- [IDistributedCache.Refresh](#)

If the actual value needs to be updated, you'd have to delete the value and then re-add it.

Delete values

To delete values in the distributed cache, call one of the remove APIs:

- [IDistributedCache.RemoveAsync](#)
- [IDistributedCache.Remove](#)

💡 Tip

While there are synchronous versions of the aforementioned APIs, please consider the fact that implementations of distributed caches are reliant on network I/O. For

this reason, it is preferred more often than not to use the asynchronous APIs.

See also

- [Dependency injection in .NET](#)
- [.NET Generic Host](#)
- [Worker Services in .NET](#)
- [Azure for .NET developers](#)
- [Cache in-memory in ASP.NET Core](#)
- [Distributed caching in ASP.NET Core](#)

 [Collaborate with us on GitHub](#)

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

 .NET

.NET feedback

.NET is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

System.Threading.Channels library

Article • 06/27/2023

The [System.Threading.Channels](#) namespace provides a set of synchronization data structures for passing data between producers and consumers asynchronously. The library targets .NET Standard and works on all .NET implementations.

This library is available in the [System.Threading.Channels](#) NuGet package. However, if you're using .NET Core 3.0 or later, the package is included as part of the framework.

Producer/consumer conceptual programming model

Channels are an implementation of the producer/consumer conceptual programming model. In this programming model, producers asynchronously produce data, and consumers asynchronously consume that data. In other words, this model hands off data from one party to another. Try to think of channels as you would any other common generic collection type, such as a `List<T>`. The primary difference is that this collection manages synchronization and provides various consumption models through factory creation options. These options control the behavior of the channels, such as how many elements they're allowed to store and what happens if that limit is reached, or whether the channel is accessed by multiple producers or multiple consumers concurrently.

Bounding strategies

Depending on how a `Channel<T>` is created, its reader and writer behave differently.

To create a channel that specifies a maximum capacity, call [Channel.CreateBounded](#). To create a channel that is used by any number of readers and writers concurrently, call [Channel.CreateUnbounded](#). Each bounding strategy exposes various creator-defined options, either [BoundedChannelOptions](#) or [UnboundedChannelOptions](#) respectively.

ⓘ Note

Regardless of the bounding strategy, a channel will always throw a [ChannelClosedException](#) when it's used after it's been closed.

Unbounded channels

To create an unbounded channel, call one of the [Channel.CreateUnbounded](#) overloads:

C#

```
var channel = Channel.CreateUnbounded<T>();
```

When you create an unbounded channel, by default, the channel can be used by any number of readers and writers concurrently. Alternatively, you can specify nondefault behavior when creating an unbounded channel by providing an [UnboundedChannelOptions](#) instance. The channel's capacity is unbounded and all writes are performed synchronously. For more examples, see [Unbounded creation patterns](#).

Bounded channels

To create a bounded channel, call one of the [Channel.CreateBounded](#) overloads:

C#

```
var channel = Channel.CreateBounded<T>(7);
```

The preceding code creates a channel that has a maximum capacity of 7 items. When you create a bounded channel, the channel is bound to a maximum capacity. When the bound is reached, the default behavior is that the channel asynchronously blocks the producer until space becomes available. You can configure this behavior by specifying an option when you create the channel. Bounded channels can be created with any capacity value greater than zero. For other examples, see [Bounded creation patterns](#).

Full mode behavior

When using a bounded channel, you can specify the behavior the channel adheres to when the configured bound is reached. The following table lists the full mode behaviors for each [BoundedChannelFullMode](#) value:

| Value | Behavior |
|---|--|
| BoundedChannelFullMode.Wait | This is the default value. Calls to <code>WriteAsync</code> wait for space to be available in order to complete the write operation. Calls to <code>TryWrite</code> return <code>false</code> immediately. |
| BoundedChannelFullMode.DropNewest | Removes and ignores the newest item in the channel in |

| Value | Behavior |
|-----------------------------------|--|
| | order to make room for the item being written. |
| BoundedChannelFullMode.DropOldest | Removes and ignores the oldest item in the channel in order to make room for the item being written. |
| BoundedChannelFullMode.DropWrite | Drops the item being written. |

ⓘ Important

Whenever a [Channel<TWrite,TRead>.Writer](#) produces faster than a [Channel<TWrite,TRead>.Reader](#) can consume, the channel's writer experiences back pressure.

Producer APIs

The producer functionality is exposed on the [Channel<TWrite,TRead>.Writer](#). The producer APIs and expected behavior are detailed in the following table:

| API | Expected behavior |
|---|--|
| ChannelWriter<T>.Complete | Marks the channel as being complete, meaning no more items are written to it. |
| ChannelWriter<T>.TryComplete | Attempts to mark the channel as being completed, meaning no more data is written to it. |
| ChannelWriter<T>.TryWrite | Attempts to write the specified item to the channel. When used with an unbounded channel, this always returns <code>true</code> unless the channel's writer signals completion with either ChannelWriter<T>.Complete , or ChannelWriter<T>.TryComplete . |
| ChannelWriter<T>.WaitToWriteAsync | Returns a ValueTask<TResult> that completes when space is available to write an item. |
| ChannelWriter<T>.WriteAsync | Asynchronously writes an item to the channel. |

Consumer APIs

The consumer functionality is exposed on the [Channel<TWrite,TRead>.Reader](#). The consumer APIs and expected behavior are detailed in the following table:

| API | Expected behavior |
|---|--|
| <code>ChannelReader<T>.ReadAllAsync</code> | Creates an <code>IAsyncEnumerable<T></code> that enables reading all of the data from the channel. |
| <code>ChannelReader<T>.ReadAsync</code> | Asynchronously reads an item from the channel. |
| <code>ChannelReader<T>.TryPeek</code> | Attempts to peek at an item from the channel. |
| <code>ChannelReader<T>.TryRead</code> | Attempts to read an item from the channel. |
| <code>ChannelReader<T>.WaitToReadAsync</code> | Returns a <code>ValueTask<TResult></code> that completes when data is available to read. |

Common usage patterns

There are several usage patterns for channels. The API is designed to be simple, consistent, and as flexible as possible. All of the asynchronous methods return a `ValueTask` (or `ValueTask<bool>`) that represents a lightweight asynchronous operation that can avoid allocating if the operation completes synchronously and potentially even asynchronously. Additionally, the API is designed to be composable, in that the creator of a channel makes promises about its intended usage. When a channel is created with certain parameters, the internal implementation can operate more efficiently knowing these promises.

Creation patterns

Imagine that you're creating a producer/consumer solution for a global position system (GPS). You want to track the coordinates of a device over time. A sample coordinates object might look like this:

C#

```
/// <summary>
/// A representation of a device's coordinates,
/// which includes latitude and longitude.
/// </summary>
/// <param name="DeviceId">A unique device identifier.</param>
/// <param name="Latitude">The latitude of the device.</param>
/// <param name="Longitude">The longitude of the device.</param>
public readonly record struct Coordinates(
    Guid DeviceId,
    double Latitude,
    double Longitude);
```

Unbounded creation patterns

One common usage pattern is to create a default unbounded channel:

```
C#
```

```
var channel = Channel.CreateUnbounded<Coordinates>();
```

But instead, let's imagine that you want to create an unbounded channel with multiple producers and consumers:

```
C#
```

```
var channel = Channel.CreateUnbounded<Coordinates>(
    new UnboundedChannelOptions
    {
        SingleWriter = false,
        SingleReader = false,
        AllowSynchronousContinuations = true
    });

```

In this case, all writes are synchronous, even the `WriteAsync`. This is because an unbounded channel always has available room for a write effectively immediately. However, with `AllowSynchronousContinuations` set to `true`, the writes may end up doing work associated with a reader by executing their continuations. This doesn't affect the synchronicity of the operation.

Bounded creation patterns

With bounded channels, the configurability of the channel should be known to the consumer to help ensure proper consumption. That is, the consumer should know what behavior the channel exhibits when the configured bound is reached. Let's explore some of the common bounded creation patterns.

The simplest way to create a bounded channel is to specify a capacity:

```
C#
```

```
var channel = Channel.CreateBounded<Coordinates>(1);
```

The preceding code creates a bounded channel with a max capacity of `1`. Other options are available, some options are the same as an unbounded channel, while others are specific to bounded channels:

C#

```
var channel = Channel.CreateBounded<Coordinates>(
    new BoundedChannelOptions(1_000)
{
    SingleWriter = true,
    SingleReader = false,
    AllowSynchronousContinuations = false,
    FullMode = BoundedChannelFullMode.DropWrite
});
```

In the preceding code, the channel is created as a bounded channel that's limited to 1,000 items, with a single writer but many readers. Its full mode behavior is defined as `DropWrite`, which means that it drops the item being written if the channel is full.

To observe items that are dropped when using bounded channels, register an `itemDropped` callback:

C#

```
var channel = Channel.CreateBounded(
    new BoundedChannelOptions(10)
{
    AllowSynchronousContinuations = true,
    FullMode = BoundedChannelFullMode.DropOldest
},
static void (Coordinates dropped) =>
    Console.WriteLine($"Coordinates dropped: {dropped}"));
```

Whenever the channel is full and a new item is added, the `itemDropped` callback is invoked. In this example, the provided callback writes the item to the console, but you're free to take any other action you want.

Producer patterns

Imagine that the producer in this scenario is writing new coordinates to the channel. The producer can do this by calling `TryWrite`:

C#

```
static void ProduceWithWhileAndTryWrite(
    ChannelWriter<Coordinates> writer, Coordinates coordinates)
{
    while (coordinates is { Latitude: < 90, Longitude: < 180 })
    {
        var tempCoordinates = coordinates with
        {
```

```

        Latitude = coordinates.Latitude + .5,
        Longitude = coordinates.Longitude + 1
    };

    if (writer.TryWrite(item: tempCoordinates))
    {
        coordinates = tempCoordinates;
    }
}
}

```

The preceding producer code:

- Accepts the `channel<Coordinates>.Writer` (`ChannelWriter<Coordinates>`) as an argument, along with the initial `Coordinates`.
- Defines a conditional `while` loop that attempts to move the coordinates using `TryWrite`.

An alternative producer might use the `WriteAsync` method:

C#

```

static async ValueTask ProduceWithWhileWriteAsync(
    ChannelWriter<Coordinates> writer, Coordinates coordinates)
{
    while (coordinates is { Latitude: < 90, Longitude: < 180 })
    {
        await writer.WriteAsync(
            item: coordinates = coordinates with
            {
                Latitude = coordinates.Latitude + .5,
                Longitude = coordinates.Longitude + 1
            });
    }

    writer.Complete();
}

```

Again, the `ChannelWriter<Coordinates>.Writer` is used within a `while` loop. But this time, the `WriteAsync` method is called. The method will continue only after the coordinates have been written. When the `while` loop exits, a call to `Complete` is made, which signals that no more data is written to the channel.

Another producer pattern is to use the `WaitToWriteAsync` method, consider the following code:

C#

```

static async ValueTask ProduceWithWaitToWriteAsync(
    ChannelWriter<Coordinates> writer, Coordinates coordinates)
{
    while (coordinates is { Latitude: < 90, Longitude: < 180 } &&
        await writer.WaitToWriteAsync())
    {
        var tempCoordinates = coordinates with
        {
            Latitude = coordinates.Latitude + .5,
            Longitude = coordinates.Longitude + 1
        };

        if (writer.TryWrite(item: tempCoordinates))
        {
            coordinates = tempCoordinates;
        }

        await Task.Delay(TimeSpan.FromMilliseconds(10));
    }

    writer.Complete();
}

```

As part of the conditional `while`, the result of the `WaitToWriteAsync` call is used to determine whether to continue the loop.

Consumer patterns

There are several common channel consumer patterns. When a channel is never ending, meaning it produces data indefinitely, the consumer could use a `while (true)` loop, and read data as it becomes available:

C#

```

static async ValueTask ConsumeWithWhileAsync(
    ChannelReader<Coordinates> reader)
{
    while (true)
    {
        // May throw ChannelClosedException if
        // the parent channel's writer signals complete.
        Coordinates coordinates = await reader.ReadAsync();
        Console.WriteLine(coordinates);
    }
}

```

 Note

This code will throw an exception if the channel is closed.

An alternative consumer could avoid this concern by using a nested while loop, as shown in the following code:

C#

```
static async ValueTask ConsumeWithNestedWhileAsync(
    ChannelReader<Coordinates> reader)
{
    while (await reader.WaitToReadAsync())
    {
        while (reader.TryRead(out Coordinates coordinates))
        {
            Console.WriteLine(coordinates);
        }
    }
}
```

In the preceding code, the consumer waits to read data. Once the data is available, the consumer tries to read it. These loops continue to evaluate until the producer of the channel signals that it no longer has data to be read. With that said, when a producer is known to have a finite number of items it produces and it signals completion, the consumer can use `await foreach` semantics to iterate over the items:

C#

```
static async ValueTask ConsumeWithAwaitForeachAsync(
    ChannelReader<Coordinates> reader)
{
    await foreach (Coordinates coordinates in reader.ReadAllAsync())
    {
        Console.WriteLine(coordinates);
    }
}
```

The preceding code uses the `ReadAllAsync` method to read all of the coordinates from the channel.

See also

- [On .NET show: Working with Channels in .NET](#)
- [.NET Blog: An Introduction to System.Threading.Channels](#) ↗
- [Managed threading basics](#)

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

System.MidpointRounding enum

Article • 01/08/2024

This article provides supplementary remarks to the reference documentation for this API.

Use the [MidpointRounding](#) enumeration with appropriate overloads of [Math.Round](#), [MathF.Round](#), and [Decimal.Round](#) to provide more control of the rounding process.

There are two overall rounding strategies—*round to nearest* and *directed rounding*—and each enumeration field participates in exactly one of these strategies.

Round to nearest

Fields:

- [AwayFromZero](#)
- [ToEven](#)

A round-to-nearest operation takes an original number with an implicit or specified precision; examines the next digit, which is at that precision plus one; and returns the nearest number with the same precision as the original number. For positive numbers, if the next digit is from 0 through 4, the nearest number is toward negative infinity. If the next digit is from 6 through 9, the nearest number is toward positive infinity. For negative numbers, if the next digit is from 0 through 4, the nearest number is toward positive infinity. If the next digit is from 6 through 9, the nearest number is toward negative infinity.

If the next digit is from 0 through 4 or 6 through 9, the [MidpointRounding.AwayFromZero](#) and [MidpointRounding.ToEven](#) do not affect the result of the rounding operation.

However, if the next digit is 5, which is the midpoint between two possible results, and all remaining digits are zero or there are no remaining digits, the nearest number is ambiguous. In this case, the round-to-nearest modes in [MidpointRounding](#) enable you to specify whether the rounding operation returns the nearest number away from zero or the nearest even number.

The following table demonstrates the results of rounding some negative and positive numbers in conjunction with round-to-nearest modes. The precision used to round the numbers is zero, which means the number after the decimal point affects the rounding operation. For example, for the number -2.5, the digit after the decimal point is 5. Because that digit is the midpoint, you can use a [MidpointRounding](#) value to determine

the result of rounding. If `AwayFromZero` is specified, -3 is returned because it is the nearest number away from zero with a precision of zero. If `ToEven` is specified, -2 is returned because it is the nearest even number with a precision of zero.

[+] Expand table

| Original number | AwayFromZero | ToEven |
|-----------------|--------------|--------|
| 3.5 | 4 | 4 |
| 2.8 | 3 | 3 |
| 2.5 | 3 | 2 |
| 2.1 | 2 | 2 |
| -2.1 | -2 | -2 |
| -2.5 | -3 | -2 |
| -2.8 | -3 | -3 |
| -3.5 | -4 | -4 |

Directed rounding

Fields:

- `ToNegativeInfinity`
- `ToPositiveInfinity`
- `ToZero`

A directed-rounding operation takes an original number with an implicit or specified precision and returns the next closest number in a specific direction with the same precision as the original number. Directed modes on `MidpointRounding` control toward which predefined number the rounding is performed.

The following table demonstrates the results of rounding some negative and positive numbers in conjunction with directed-rounding modes. The precision used to round the numbers is zero, which means the number before the decimal point is affected by the rounding operation.

[+] Expand table

| Original number | ToNegativeInfinity | ToPositiveInfinity | ToZero |
|-----------------|--------------------|--------------------|--------|
| 3.5 | 3 | 4 | 3 |
| 2.8 | 2 | 3 | 2 |
| 2.5 | 2 | 3 | 2 |
| 2.1 | 2 | 3 | 2 |
| -2.1 | -3 | -2 | -2 |
| -2.5 | -3 | -2 | -2 |
| -2.8 | -3 | -2 | -2 |
| -3.5 | -4 | -3 | -3 |

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



.NET feedback

.NET is an open source project.
Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

The Microsoft.Win32.Registry class

Article • 01/04/2024

This article provides supplementary remarks to the reference documentation for this API.

The [Registry](#) class provides the set of standard root keys found in the registry on machines running Windows. The registry is a storage facility for information about applications, users, and default system settings. Applications can use the registry for storing information that needs to be preserved after the application is closed, and access that same information when the application is reloaded. For example, you can store color preferences, screen locations, or the size of a window. You can control this data for each user by storing the information in a different location in the registry.

The base, or root, [RegistryKey](#) instances that are exposed by the [Registry](#) class delineate the basic storage mechanism for subkeys and values in the registry. All keys are read-only because the registry depends on their existence. The keys exposed by [Registry](#) are:

[+] Expand table

| Key | Description |
|---------------------------------|--|
| CurrentUser | Stores information about user preferences. |
| LocalMachine | Stores configuration information for the local machine. |
| ClassesRoot | Stores information about types (and classes) and their properties. |
| Users | Stores information about the default user configuration. |
| PerformanceData | Stores performance information for software components. |
| CurrentConfig | Stores non-user-specific hardware information. |
| DynData | Stores dynamic data. |

Once you've identified the root key under which you want to store/retrieve information from the registry, you can use the [RegistryKey](#) class to add or remove subkeys and manipulate the values for a given key.

Hardware devices can place information in the registry automatically using the Plug and Play interface. Software for installing device drivers can place information in the registry by writing to standard APIs.

Static methods for getting and setting values

The [Registry](#) class also contains `static GetValue` and `SetValue` methods for setting and retrieving values from registry keys. These methods open and close registry keys each time they're used. So when you access a large number of values, they don't perform as well as analogous methods in the [RegistryKey](#) class.

The [RegistryKey](#) class also provides methods that allow you to:

- Set Windows access control security for registry keys.
- Test the data type of a value before retrieving it.
- Delete keys.

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

 .NET

.NET feedback

.NET is an open source project.
Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

System.Uri class

Article • 01/08/2024

This article provides supplementary remarks to the reference documentation for this API.

A URI is a compact representation of a resource available to your application on the intranet or internet. The [Uri](#) class defines the properties and methods for handling URIs, including parsing, comparing, and combining. The [Uri](#) class properties are read-only; to create a modifiable object, use the [UriBuilder](#) class.

Relative URLs (for example, "/new/index.htm") must be expanded with respect to a base URI so that they are absolute. The [MakeRelativeUri](#) method is provided to convert absolute URLs to relative URLs when necessary.

The [Uri](#) constructors do not escape URI strings if the string is a well-formed URI including a scheme identifier.

The [Uri](#) properties return a canonical data representation in escaped encoding, with all characters with Unicode values greater than 127 replaced with their hexadecimal equivalents. To put the URI in canonical form, the [Uri](#) constructor performs the following steps:

- Converts the URI scheme to lowercase.
- Converts the host name to lowercase.
- If the host name is an IPv6 address, the canonical IPv6 address is used. Scopeld and other optional IPv6 data are removed.
- Removes default and empty port numbers.
- Converts implicit file paths without the file:// scheme (for example, "C:\my\file") to explicit file paths with the file:// scheme.
- Escaped characters (also known as percent-encoded octets) that don't have a reserved purpose are decoded (also known as being unescaped). These unreserved characters include uppercase and lowercase letters (%41-%5A and %61-%7A), decimal digits (%30-%39), hyphen (%2D), period (%2E), underscore (%5F), and tilde (%7E).
- Canonicalizes the path for hierarchical URIs by compacting sequences such as ./, ../, and // (whether or not the sequence is escaped). Note that there are some schemes for which these sequences are not compacted.

- For hierarchical URIs, if the host is not terminated with a forward slash (/), one is added.
- By default, any reserved characters in the URI are escaped in accordance with RFC 2396. This behavior changes if International Resource Identifiers or International Domain Name parsing is enabled in which case reserved characters in the URI are escaped in accordance with RFC 3986 and RFC 3987.

As part of canonicalization in the constructor for some schemes, dot-segments and empty segments (./., /.../, and //) are compacted (in other words, they are removed). The schemes for which [Uri](#) compacts segments include http, https, tcp, net.pipe, and net.tcp. For some other schemes, these sequences are not compacted. The following code snippet shows how compacting looks in practice. The escaped sequences are unescaped, if necessary, and then compacted.

C#

```
var uri = new Uri("http://myUrl/../../"); // http scheme, unescaped
OR
var uri = new Uri("http://myUrl/%2E%2E/%2E%2E"); // http scheme, escaped
OR
var uri = new Uri("ftp://myUrl/../../"); // ftp scheme, unescaped
OR
var uri = new Uri("ftp://myUrl/%2E%2E/%2E%2E"); // ftp scheme, escaped

Console.WriteLine($"AbsoluteUri: {uri.AbsoluteUri}");
Console.WriteLine($"PathAndQuery: {uri.PathAndQuery}");
```

When this code executes, it returns output similar to the following text.

Output

```
AbsoluteUri: http://myurl/
PathAndQuery: /
```

You can transform the contents of the [Uri](#) class from an escape encoded URI reference to a readable URI reference by using the [ToString](#) method. Note that some reserved characters might still be escaped in the output of the [ToString](#) method. This is to support unambiguous reconstruction of a URI from the value returned by [ToString](#).

Some URIs include a fragment identifier or a query or both. A fragment identifier is any text that follows a number sign (#), not including the number sign; the fragment text is stored in the [Fragment](#) property. Query information is any text that follows a question mark (?) in the URI; the query text is stored in the [Query](#) property.

Note

The `URI` class supports the use of IP addresses in both quad-notation for IPv4 protocol and colon-hexadecimal for IPv6 protocol. Remember to enclose the IPv6 address in square brackets, as in `http://[::1]`.

International resource identifier support

Web addresses are typically expressed using uniform resource identifiers that consist of a very restricted set of characters:

- Upper and lower case ASCII letters from the English alphabet.
- Digits from 0 to 9.
- A small number of other ASCII symbols.

The specifications for URIs are documented in RFC 2396, RFC 2732, RFC 3986, and RFC 3987 published by the Internet Engineering Task Force (IETF).

With the growth of the Internet, there is a growing need to identify resources using languages other than English. Identifiers which facilitate this need and allow non-ASCII characters (characters in the Unicode/ISO 10646 character set) are known as International Resource Identifiers (IRIs). The specifications for IRIs are documented in RFC 3987 published by IETF. Using IRIs allows a URL to contain Unicode characters.

The existing `Uri` class was extended in .NET Framework v3.5, 3.0 SP1, and 2.0 SP1 to provide IRI support based on RFC 3987. Users of .NET Framework versions before version 4.5 will not see any change from .NET Framework 2.0 behavior unless they specifically enable IRI. This ensures application compatibility with prior versions of .NET Framework.

In .NET Framework 4.5 and later versions, IRI is always enabled and can't be changed using a configuration option. To enable support for IRI in .NET Framework versions prior to .NET Framework 4.5, set a configuration option in the `machine.config` or in the `app.config` file. Specify whether you want Internationalized Domain Name (IDN) parsing applied to the domain name and whether IRI parsing rules should be applied. For example:

XML

```
<configuration>
  <uri>
    <idn enabled="All" />
    <iriParsing enabled="true" />
```

```
</uri>  
</configuration>
```

Enabling IDN converts all Unicode labels in a domain name to their Punycode equivalents. Punycode names contain only ASCII characters and always start with the xn- prefix. The reason for this is to support existing DNS servers on the Internet, since most DNS servers only support ASCII characters (see RFC 3940).

Enabling IRI and IDN affects the value of the [Uri.DnsSafeHost](#) property. Enabling IRI and IDN can also change the behavior of the [Equals](#), [OriginalString](#), [GetComponents](#), and [IsWellFormedOriginalString](#) methods.

There are three possible values for IDN depending on the DNS servers that are used:

- idn enabled = All

This value will convert any Unicode domain names to their Punycode equivalents (IDN names).

- idn enabled = AllExceptIntranet

This value will convert all Unicode domain names not on the local Intranet to use the Punycode equivalents (IDN names). In this case to handle international names on the local Intranet, the DNS servers that are used for the Intranet should support Unicode name resolution.

- idn enabled = None

This value will not convert any Unicode domain names to use Punycode. This is the default value that's consistent with .NET Framework 2.0 behavior.

When IRI parsing is enabled (`iriParsing enabled = true`), normalization and character checking are done according to the latest IRI rules in RFC 3986 and RFC 3987. When IRI parsing is disabled, normalization and character checking are performed according to RFC 2396 and RFC 2732 (for IPv6 literals). In versions of .NET Framework before version 4.5, the default value is `false`. In .NET Framework 4.5+, .NET Core, and .NET 5+, the default value is `true`, and the enabled state of IRI parsing cannot be modified by settings in a `.config` file.

IRI and IDN processing in the [Uri](#) class can also be controlled using the [System.Configuration.IriParsingElement](#), [System.Configuration.IdnElement](#), and [System.Configuration.UriSection](#) configuration setting classes. The [System.Configuration.IriParsingElement](#) setting enables or disables IRI processing in the [Uri](#) class. The [System.Configuration.IdnElement](#) setting enables or disables IDN

processing in the [Uri](#) class. The [System.Configuration.IriParsingElement](#) setting also indirectly controls IDN. IRI processing must be enabled for IDN processing to be possible. If IRI processing is disabled, then IDN processing will be set to the default setting where .NET Framework 2.0 behavior is used for compatibility and IDN names are not used.

The configuration setting for the [System.Configuration.IriParsingElement](#) and [System.Configuration.IdnElement](#) are read once when the first [System.Uri](#) class is constructed. Changes to configuration settings after that time are ignored.

The [System.GenericUriParser](#) class has also been extended to allow creating a customizable parser that supports IRI and IDN. The behavior of a [System.GenericUriParser](#) object is specified by passing a bitwise combination of the values available in the [System.GenericUriParserOptions](#) enumeration to the [System.GenericUriParser](#) constructor. The [GenericUriParserOptions.IriParsing](#) type indicates the parser supports the parsing rules specified in RFC 3987 for International Resource Identifiers (IRI). Whether IRI is used is dictated by the configuration values previously discussed.

The [GenericUriParserOptions.Idn](#) type indicates that the parser supports Internationalized Domain Name (IDN) parsing of host names. In .NET 5 and later versions (including .NET Core) and .NET Framework 4.5+, IDN is always used. In previous versions, a configuration option determines whether IDN is used.

Implicit file path support

[Uri](#) can also be used to represent local file system paths. These paths can be represented *explicitly* in URIs that begin with the file:// scheme, and *implicitly* in URIs that do not have the file:// scheme. As a concrete example, the following two URIs are both valid, and represent the same file path:

C#

```
Uri uri1 = new Uri("C:/test/path/file.txt") // Implicit file path.  
Uri uri2 = new Uri("file:///C:/test/path/file.txt") // Explicit file path.
```

These implicit file paths are not compliant with the URI specification and so should be avoided when possible. When using .NET Core on Unix-based systems, implicit file paths can be especially problematic, because an absolute implicit file path is *indistinguishable* from a relative path. When such ambiguity is present, [Uri](#) default to interpreting the path as an absolute URI.

Security considerations

Because of security concerns, your application should use caution when accepting `Uri` instances from untrusted sources and with `dontEscape` set to `true` in the [constructor](#).

You can check a URL string for validity by calling the [IsWellFormedOriginalString](#) method.

When dealing with untrusted user input, confirm assumptions about the newly created `Uri` instance before trusting its properties. This can be done in the following way:

C#

```
string userInput = ...;

Uri baseUri = new Uri("https://myWebsite/files/");

if (!Uri.TryCreate(baseUri, userInput, out Uri newUri))
{
    // Fail: invalid input.
}

if (!baseUri.IsBaseOf(newUri))
{
    // Fail: the Uri base has been modified - the created Uri is not rooted
    // in the original directory.
}
```

This validation can be used in other cases, like when dealing with UNC paths, by simply changing the `baseUri`:

C#

```
Uri baseUri = new Uri(@"\\host\share\some\directory\name\");
```

Performance considerations

If you use a `Web.config` file that contains URLs to initialize your application, additional time is required to process the URLs if their scheme identifiers are nonstandard. In such a case, initialize the affected parts of your application when the URLs are needed, not at start time.



Collaborate with us on
GitHub

.NET

.NET feedback

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET is an open source project.
Select a link to provide feedback:

-  [Open a documentation issue](#)
-  [Provide product feedback](#)

System.Type class

Article • 01/08/2024

This article provides supplementary remarks to the reference documentation for this API.

The [Type](#) class is the root of the [System.Reflection](#) functionality and is the primary way to access metadata. Use the members of [Type](#) to get information about a type declaration, about the members of a type (such as the constructors, methods, fields, properties, and events of a class), as well as the module and the assembly in which the class is deployed.

No permissions are required for code to use reflection to get information about types and their members, regardless of their access levels. No permissions are required for code to use reflection to access public members, or other members whose access levels would make them visible during normal compilation. However, in order for your code to use reflection to access members that would normally be inaccessible, such as private or internal methods, or protected fields of a type your class does not inherit, your code must have [ReflectionPermission](#). See [Security Considerations for Reflection](#).

`Type` is an abstract base class that allows multiple implementations. The system will always provide the derived class `RuntimeType`. In reflection, all classes beginning with the word Runtime are created only once per object in the system and support comparison operations.

ⓘ Note

In multithreading scenarios, do not lock `Type` objects in order to synchronize access to `static` data. Other code, over which you have no control, might also lock your class type. This might result in a deadlock. Instead, synchronize access to static data by locking a private `static` object.

ⓘ Note

A derived class can access protected members of the calling code's base classes. Also, access is allowed to assembly members of the calling code's assembly. As a rule, if you are allowed access in early-bound code, then you are also allowed access in late-bound code.

ⓘ Note

Interfaces that extend other interfaces do not inherit the methods defined in the extended interfaces.

What types does a Type object represent?

This class is thread safe; multiple threads can concurrently read from an instance of this type. An instance of the [Type](#) class can represent any of the following types:

- Classes
- Value types
- Arrays
- Interfaces
- Enumerations
- Delegates
- Constructed generic types and generic type definitions
- Type arguments and type parameters of constructed generic types, generic type definitions, and generic method definitions

Retrieve a Type object

The [Type](#) object associated with a particular type can be obtained in the following ways:

- The instance [Object.GetType](#) method returns a [Type](#) object that represents the type of an instance. Because all managed types derive from [Object](#), the [GetType](#) method can be called on an instance of any type.

The following example calls the [Object.GetType](#) method to determine the runtime type of each object in an object array.

C#

```
object[] values = { "word", true, 120, 136.34, 'a' };
foreach (var value in values)
    Console.WriteLine("{0} - type {1}", value,
                      value.GetType().Name);

// The example displays the following output:
//      word - type String
//      True - type Boolean
//      120 - type Int32
```

```
//      136.34 - type Double  
//      a - type Char
```

- The static [Type.GetType](#) methods return a [Type](#) object that represents a type specified by its fully qualified name.
- The [Module.GetTypes](#), [Module.GetType](#), and [Module.FindTypes](#) methods return [Type](#) objects that represent the types defined in a module. The first method can be used to obtain an array of [Type](#) objects for all the public and private types defined in a module. (You can obtain an instance of [Module](#) through the [Assembly.GetModule](#) or [Assembly.GetModules](#) method, or through the [Type.Module](#) property.)
- The [System.Reflection.Assembly](#) object contains a number of methods to retrieve the classes defined in an assembly, including [Assembly.GetType](#), [Assembly.GetTypes](#), and [Assembly.GetExportedTypes](#).
- The [FindInterfaces](#) method returns a filtered list of interface types supported by a type.
- The [GetElementType](#) method returns a [Type](#) object that represents the element.
- The [GetInterfaces](#) and [GetInterface](#) methods return [Type](#) objects representing the interface types supported by a type.
- The [GetTypeArray](#) method returns an array of [Type](#) objects representing the types specified by an arbitrary set of objects. The objects are specified with an array of type [Object](#).
- The [GetTypeFromProgID](#) and [GetTypeFromCLSID](#) methods are provided for COM interoperability. They return a [Type](#) object that represents the type specified by a [ProgID](#) or [CLSID](#).
- The [GetTypeFromHandle](#) method is provided for interoperability. It returns a [Type](#) object that represents the type specified by a class handle.
- The C# [typeof](#) operator, the C++ [typeid](#) operator, and the Visual Basic [GetType](#) operator obtain the [Type](#) object for a type.
- The [MakeGenericType](#) method returns a [Type](#) object representing a constructed generic type, which is an open constructed type if its [ContainsGenericParameters](#) property returns [true](#), and a closed constructed type otherwise. A generic type can be instantiated only if it is closed.

- The [MakeArrayType](#), [MakePointerType](#), and [MakeByRefType](#) methods return [Type](#) objects that represent, respectively, an array of a specified type, a pointer to a specified type, and the type of a reference parameter (`ref` in C#, 'byref' in F#, `ByRef` in Visual Basic).

Compare type objects for equality

A [Type](#) object that represents a type is unique; that is, two [Type](#) object references refer to the same object if and only if they represent the same type. This allows for comparison of [Type](#) objects using reference equality. The following example compares the [Type](#) objects that represent a number of integer values to determine whether they are of the same type.

C#

```
long number1 = 1635429;
int number2 = 16203;
double number3 = 1639.41;
long number4 = 193685412;

// Get the type of number1.
Type t = number1.GetType();

// Compare types of all objects with number1.
Console.WriteLine("Type of number1 and number2 are equal: {0}",
                  Object.ReferenceEquals(t, number2.GetType()));
Console.WriteLine("Type of number1 and number3 are equal: {0}",
                  Object.ReferenceEquals(t, number3.GetType()));
Console.WriteLine("Type of number1 and number4 are equal: {0}",
                  Object.ReferenceEquals(t, number4.GetType()));

// The example displays the following output:
//      Type of number1 and number2 are equal: False
//      Type of number1 and number3 are equal: False
//      Type of number1 and number4 are equal: True
```

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For

.NET

.NET feedback

.NET is an open source project.
Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

more information, see [our contributor guide](#).

System.Drawing.Drawing2D.Matrix class

Article • 01/08/2024

This article provides supplementary remarks to the reference documentation for this API.

The [Matrix](#) class encapsulates a 3-by-3 affine matrix that represents a geometric transform.

In GDI+, you can store an affine transformation in a [Matrix](#) object. Because the third column of a matrix that represents an affine transformation is always (0, 0, 1), you specify only the six numbers in the first two columns when you construct a [Matrix](#) object. The statement `Matrix myMatrix = new Matrix(0, 1, -1, 0, 3, 4)` constructs the matrix shown in the following figure.

The diagram shows a 3x3 matrix with dashed lines indicating its structure. The top-left 2x2 block is labeled 'Linear part' with an arrow. The bottom-right 2x2 block is labeled 'Translation part' with an arrow. The matrix elements are:

$$\begin{bmatrix} 0 & 1 & 0 \\ -1 & 0 & 0 \\ 3 & 4 & 1 \end{bmatrix}$$

ⓘ Note

In .NET 6 and later versions, the [System.Drawing.Common package](#), which includes this type, is only supported on Windows operating systems. Use of this type in cross-platform apps causes compile-time warnings and run-time exceptions. For more information, see [System.Drawing.Common only supported on Windows](#).

Composite transformations

A composite transformation is a sequence of transformations, one followed by the other. Consider the matrices and transformations in the following list:

expand Expand table

| Matrix | Transformation |
|----------|-------------------|
| Matrix A | Rotate 90 degrees |

| Matrix | Transformation |
|----------|---|
| Matrix B | Scale by a factor of 2 in the x direction |
| Matrix C | Translate 3 units in the y direction |

If you start with the point (2, 1) - represented by the matrix [2 1 1] - and multiply by A, then B, then C, the point (2, 1) will undergo the three transformations in the order listed.

$$[2 \ 1 \ 1]ABC = [-2 \ 5 \ 1]$$

Rather than store the three parts of the composite transformation in three separate matrices, you can multiply A, B, and C together to get a single 3×3 matrix that stores the entire composite transformation. Suppose $ABC = D$. Then a point multiplied by D gives the same result as a point multiplied by A, then B, then C.

$$[2 \ 1 \ 1]D = [-2 \ 5 \ 1]$$

The following illustration shows the matrices A, B, C, and D.

$$\begin{bmatrix} 0 & 1 & 0 \\ -1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 2 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 3 & 1 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 \\ -2 & 0 & 0 \\ 0 & 3 & 1 \end{bmatrix}$$

A B C = D

The fact that the matrix of a composite transformation can be formed by multiplying the individual transformation matrices means that any sequence of affine transformations can be stored in a single [Matrix](#) object.

⊗ Caution

The order of a composite transformation is important. In general, rotate, then scale, then translate is not the same as scale, then rotate, then translate. Similarly, the order of matrix multiplication is important. In general, ABC is not the same as BAC.

The [Matrix](#) class provides several methods for building a composite transformation: [Multiply](#), [Rotate](#), [RotateAt](#), [Scale](#), [Shear](#), and [Translate](#). The following example creates the matrix of a composite transformation that first rotates 30 degrees, then scales by a factor of 2 in the y direction, and then translates 5 units in the x direction:

C#

```
Matrix myMatrix = new Matrix();
myMatrix.Rotate(30);
```

```
myMatrix.Scale(1, 2, MatrixOrder.Append);  
myMatrix.Translate(5, 0, MatrixOrder.Append);
```

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

.NET is an open source project.
Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

System.Runtime.CompilerServices.InternalVisibleToAttribute class

Article • 01/08/2024

This article provides supplementary remarks to the reference documentation for this API.

The [InternalsVisibleToAttribute](#) attribute specifies that types that are ordinarily visible only within the current assembly are visible to a specified assembly.

Ordinarily, types and members with [internal scope in C#](#) or [Friend scope in Visual Basic](#) are visible only in the assembly in which they are defined. Types and members with [protected internal](#) scope ([Protected Friend](#) scope in Visual Basic) are visible only in their own assembly or to types that derive from their containing class. Types and members with [private protected](#) scope ([Private Protected](#) scope in Visual Basic) are visible in the containing class or in types that derive from their containing class within the current assembly.

The [InternalsVisibleToAttribute](#) attribute makes these types and members also visible to the types in a specified assembly, which is known as a friend assembly. This applies only to [internal](#) ([Friend](#) in Visual Basic), [protected internal](#) ([Protected Friend](#) in Visual Basic), and [private protected](#) ([Private Protected](#) in Visual Basic) members, but not [private](#) ones.

ⓘ Note

In the case of [private protected](#) ([Private Protected](#) in Visual Basic) members, the [InternalsVisibleToAttribute](#) attribute extends accessibility only to types that derive from the *containing class* of the member.

The attribute is applied at the assembly level. This means that it can be included at the beginning of a source code file, or it can be included in the AssemblyInfo file in a Visual Studio project. You can use the attribute to specify a single friend assembly that can access the internal types and members of the current assembly. You can define multiple friend assemblies in two ways. They can appear as individual assembly-level attributes, as the following example illustrates.

C#

```
[assembly:InternalsVisibleTo("Friend1a")]
[assembly:InternalsVisibleTo("Friend1b")]
```

They can also appear with separate [InternalsVisibleToAttribute](#) tags but a single `assembly` keyword, as the following example illustrates.

C#

```
[assembly:InternalsVisibleTo("Friend2a"),
    InternalsVisibleTo("Friend2b")]
```

The friend assembly is identified by the [InternalsVisibleToAttribute](#) constructor. Both the current assembly and the friend assembly must be unsigned, or both assemblies must be signed with a strong name.

If both assemblies are unsigned, the `assemblyName` argument consists of the name of the friend assembly, specified without a directory path or file name extension.

If both assemblies are signed with a strong name, the argument to the [InternalsVisibleToAttribute](#) constructor must consist of the name of the assembly without its directory path or file name extension, along with the full public key (and not its public key token). To get the full public key of a strong-named assembly, see the [Get the full public key](#) section later in this article. For more information about using [InternalsVisibleToAttribute](#) with strong-named assemblies, see the [InternalsVisibleToAttribute](#) constructor.

Do not include values for the [CultureInfo](#), [Version](#), or [ProcessorArchitecture](#) field in the argument; the Visual Basic, C#, and C++ compilers treat this as a compiler error. If you use a compiler that does not treat it as an error (such as the [IL Assembler \(ILAsm.exe\)](#)) and the assemblies are strong-named, a [MethodAccessException](#) exception is thrown the first time the specified friend assembly accesses the assembly that contains the [InternalsVisibleToAttribute](#) attribute.

For more information about how to use this attribute, see [Friend assemblies](#) and [C++ friend assemblies](#).

Get the full public key

You can use the [Strong Name Tool \(Sn.exe\)](#) to retrieve the full public key from a strong-named key (.snk) file. To do this, you perform the following steps:

1. Extract the public key from the strong-named key file to a separate file:

```
Sn -p <snk_file> <outfile>
```

2. Display the full public key to the console:

```
Sn -tp <outfile>
```

3. Copy and paste the full public key value into your source code.

Compile the friend assembly with C#

If you use the C# compiler to compile the friend assembly, you must explicitly specify the name of the output file (.exe or .dll) by using the **/out** compiler option. This is required because the compiler has not yet generated the name for the assembly it is building at the time it is binding to external references. The **/out** compiler option is optional for the Visual Basic compiler, and the corresponding **-out** or **-o** compiler option should not be used when compiling friend assemblies with the F# compiler.

Compile the friend assembly with C++

In C++, in order to make the internal members enabled by the [InternalsVisibleToAttribute](#) attribute accessible to a friend assembly, you must use the `as_friend` attribute in the C++ directive. For more information, see [Friend Assemblies \(C++\)](#).

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

.NET is an open source project.
Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

System.Runtime.Versioning.ComponentGuaranteesAttribute class

Article • 01/08/2024

This article provides supplementary remarks to the reference documentation for this API.

The [ComponentGuaranteesAttribute](#) is used by developers of components and class libraries to indicate the level of compatibility that consumers of their libraries can expect across multiple versions. It indicates the level of guarantee that a future version of the library or component will not break an existing client. Clients can then use the [ComponentGuaranteesAttribute](#) as an aid in designing their own interfaces to ensure stability across versions.

ⓘ Note

The common language runtime (CLR) does not use this attribute in any way. Its value lies in formally documenting the intent of the component author. Compile-time tools can also use these declarations to detect compile-time errors that would otherwise break the declared guarantee.

Levels of compatibility

The [ComponentGuaranteesAttribute](#) supports the following levels of compatibility, which are represented by members of the [ComponentGuaranteesOptions](#) enumeration:

- No version-to-version compatibility ([ComponentGuaranteesOptions.None](#)). The client can expect that future versions will break the existing client. For more information, see the [No compatibility](#) section later in this article.
- Side-by-side version-to-version compatibility ([ComponentGuaranteesOptions.SideBySide](#)). The component has been tested to work when more than one version of the assembly is loaded in the same application domain. In general, future versions can break compatibility. However, when breaking changes are made, the old version is not modified but exists alongside the new version. Side-by-side execution is the expected way to make existing clients work when breaking changes are made. For more information, see the [Side-by-side compatibility](#) section later in this article.

- Stable version-to-version compatibility ([ComponentGuaranteesOptions.Stable](#)). Future versions should not break the client, and side-by-side execution should not be needed. However, if the client is inadvertently broken, it may be possible to use side-by-side execution to fix the problem. For more information, see the [Stable compatibility](#) section.
- Exchange version-to-version compatibility ([ComponentGuaranteesOptions.Exchange](#)). Extraordinary care is taken to ensure that future versions will not break the client. The client should use only these types in the signature of interfaces that are used for communication with other assemblies that are deployed independently of one another. Only one version of these types is expected to be in a given application domain, which means that if a client breaks, side-by-side execution cannot fix the compatibility problem. For more information, see the [Exchange type compatibility](#) section.

The following sections discuss each level of guarantee in greater detail.

No compatibility

Marking a component as [ComponentGuaranteesOptions.None](#) indicates that the provider makes no guarantees about compatibility. Clients should avoid taking any dependencies on the exposed interfaces. This level of compatibility is useful for types that are experimental or that are publicly exposed but are intended only for components that are always updated at the same time. [None](#) explicitly indicates that this component should not be used by external components.

Side-by-side compatibility

Marking a component as [ComponentGuaranteesOptions.SideBySide](#) indicates that the component has been tested to work when more than one version of the assembly is loaded into the same application domain. Breaking changes are allowed as long as they are made to the assembly that has the greater version number. Components that are bound to an old version of the assembly are expected to continue to bind to the old version, and other components can bind to the new version. It is also possible to update a component that is declared to be [SideBySide](#) by destructively modifying the old version.

Stable compatibility

Marking a type as [ComponentGuaranteesOptions.Stable](#) indicates that the type should remain stable across versions. However, it may also be possible for side-by-side versions

of a stable type to exist in the same application domain.

Stable types maintain a high binary compatibility bar. Because of this, providers should avoid making breaking changes to stable types. The following kinds of changes are acceptable:

- Adding private instance fields to, or removing fields from, a type, as long as this does not break the serialization format.
- Changing a non-serializable type to a serializable type. (However, a serializable type cannot be changed to a non-serializable type.)
- Throwing new, more derived exceptions from a method.
- Improving the performance of a method.
- Changing the range of return values, as long as the change does not adversely affect the majority of clients.
- Fixing serious bugs, if the business justification is high and the number of adversely affected clients is low.

Because new versions of stable components are not expected to break existing clients, generally only one version of a stable component is needed in an application domain. However, this is not a requirement, because stable types are not used as well-known exchange types that all components agree upon. Therefore, if a new version of a stable component does inadvertently break some component, and if other components need the new version, it may be possible to fix the problem by loading both the old and new component.

[Stable](#) provides a stronger version compatibility guarantee than [None](#). It is a common default for multi-version components.

[Stable](#) can be combined with [SideBySide](#), which states that the component will not break compatibility but is tested to work when more than one version is loaded in a given application domain.

After a type or method is marked as [Stable](#), it can be upgraded to [Exchange](#). However, it cannot be downgraded to [None](#).

Exchange type compatibility

Marking a type as [ComponentGuaranteesOptions.Exchange](#) provides a stronger version compatibility guarantee than [Stable](#), and should be applied to the most stable of all types. These types are intended to be used for interchange between independently built components across all component boundaries in both time (any version of the CLR or any version of a component or application) and space (cross-process, cross-CLR in one

process, cross-application domain in one CLR). If a breaking change is made to an exchange type, it is impossible to fix the issue by loading multiple versions of the type.

Exchange types should be changed only when a problem is very serious (such as a severe security issue) or the probability of breakage is very low (that is, if the behavior was already broken in a random way that code could not have conceivably taken a dependency on). You can make the following kinds of changes to an exchange type:

- Add inheritance of new interface definitions.
- Add new private methods that implement the methods of newly inherited interface definitions.
- Add new static fields.
- Add new static methods.
- Add new non-virtual instance methods.

The following are considered breaking changes and are not allowed for primitive types:

- Changing serialization formats. Version-tolerant serialization is required.
- Adding or removing private instance fields. This risks changing the serialization format of the type and breaking client code that uses reflection.
- Changing the serializability of a type. A non-serializable type may not be made serializable, and vice versa.
- Throwing different exceptions from a method.
- Changing the range of a method's return values, unless the member definition raises this possibility and clearly indicates how clients should handle unknown values.
- Fixing most bugs. Consumers of the type will rely on the existing behavior.

After a component, type, or member is marked with the [Exchange](#) guarantee, it cannot be changed to either [Stable](#) or [None](#).

Typically, exchange types are the basic types (such as [Int32](#) and [String](#) in .NET) and interfaces (such as [IList<T>](#), [IEnumerable<T>](#), and [IComparable<T>](#)) that are commonly used in public interfaces.

Exchange types may publicly expose only other types that are also marked with [Exchange](#) compatibility. In addition, exchange types cannot depend on the behavior of

Windows APIs that are prone to change.

Component guarantees

The following table indicates how a component's characteristics and usage affect its compatibility guarantee.

[+] Expand table

| Component characteristics | Exchange | Stable | Side-by-Side | None |
|--|----------|--------|--------------|------|
| Can be used in interfaces between components that version independently. | Y | N | N | N |
| Can be used (privately) by an assembly that versions independently. | Y | Y | Y | N |
| Can have multiple versions in a single application domain. | N | Y | Y | Y |
| Can make breaking changes | N | N | Y | Y |
| Tested to make certain multiple versions of the assembly can be loaded together. | N | N | Y | N |
| Can make breaking changes in place. | N | N | N | Y |
| Can make very safe non-breaking servicing changes in place. | Y | Y | Y | Y |

Apply the attribute

You can apply the [ComponentGuaranteesAttribute](#) to an assembly, a type, or a type member. Its application is hierarchical. That is, by default, the guarantee defined by the [Guarantees](#) property of the attribute at the assembly level defines the guarantee of all types in the assembly and all members in those types. Similarly, if the guarantee is applied to the type, by default it also applies to each member of the type.

This inherited guarantee can be overridden by applying the [ComponentGuaranteesAttribute](#) to individual types and type members. However, guarantees that override the default can only weaken the guarantee; they cannot strengthen it. For example, if an assembly is marked with the [None](#) guarantee, its types and members have no compatibility guarantee, and any other guarantee that is applied to types or members in the assembly is ignored.

Test the guarantee

The [Guarantees](#) property returns a member of the [ComponentGuaranteesOptions](#) enumeration, which is marked with the [FlagsAttribute](#) attribute. This means that you should test for the flag that you are interested in by masking away potentially unknown flags. For example, the following example tests whether a type is marked as [Stable](#).

C#

```
// Test whether guarantee is Stable.  
if ((guarantee & ComponentGuaranteesOptions.Stable) ==  
ComponentGuaranteesOptions.Stable)  
    Console.WriteLine("{0} is marked as {1}.", typ.Name, guarantee);
```

The following example tests whether a type is marked as [Stable](#) or [Exchange](#).

C#

```
// Test whether guarantee is Stable or Exchange.  
if ((guarantee & (ComponentGuaranteesOptions.Stable |  
ComponentGuaranteesOptions.Exchange)) > 0)  
    Console.WriteLine("{0} is marked as Stable or Exchange.", typ.Name,  
guarantee);
```

The following example tests whether a type is marked as [None](#) (that is, neither [Stable](#) nor [Exchange](#)).

C#

```
// Test whether there is no guarantee (neither Stable nor Exchange).  
if ((guarantee & (ComponentGuaranteesOptions.Stable |  
ComponentGuaranteesOptions.Exchange)) == 0)  
    Console.WriteLine("{0} has no compatibility guarantee.", typ.Name,  
guarantee);
```

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For

.NET

 .NET feedback

.NET is an open source project.
Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

more information, see [our contributor guide](#).

System.Runtime.Loader.AssemblyLoadContext class

Article • 01/08/2024

This article provides supplementary remarks to the reference documentation for this API.

The [AssemblyLoadContext](#) represents a load context. Conceptually, a load context creates a scope for loading, resolving, and potentially unloading a set of assemblies.

The [AssemblyLoadContext](#) exists primarily to provide assembly loading isolation. It allows multiple versions of the same assembly to be loaded within a single process. It replaces the isolation mechanisms provided by multiple [AppDomain](#) instances in .NET Framework.

ⓘ Note

- [AssemblyLoadContext](#) does not provide any security features. All code has full permissions of the process.
- In .NET Core 2.0 - 2.2 only, [AssemblyLoadContext](#) is an abstract class. To create a concrete class in these versions, implement the [AssemblyLoadContext.Load\(AssemblyName\)](#) method.

Usage in the runtime

The runtime implements two assembly load contexts:

- [AssemblyLoadContext.Default](#) represents the runtime's default context, which is used for the application main assembly and its static dependencies.
- The [Assembly.LoadFile\(String\)](#) method isolates the assemblies it loads by instantiating the most basic [AssemblyLoadContext](#). It has a simplistic isolation scheme that loads each assembly in its own [AssemblyLoadContext](#) with no dependency resolution.

Application usage

An application can create its own [AssemblyLoadContext](#) to create a custom solution for advanced scenarios. The customization focuses on defining dependency resolution

mechanisms.

The [AssemblyLoadContext](#) provides two extension points to implement managed assembly resolution:

1. The [AssemblyLoadContext.Load\(AssemblyName\)](#) method provides the first chance for the [AssemblyLoadContext](#) to resolve, load, and return the assembly. If the [AssemblyLoadContext.Load\(AssemblyName\)](#) method returns `null`, the loader tries to load the assembly into the [AssemblyLoadContext.Default](#).
2. If the [AssemblyLoadContext.Default](#) is unable to resolve the assembly, the original [AssemblyLoadContext](#) gets a second chance to resolve the assembly. The runtime raises the [Resolving](#) event.

Additionally, the [AssemblyLoadContext.LoadUnmanagedDll\(String\)](#) virtual method allows customization of the default unmanaged assembly resolution. The default implementation returns `null`, which causes the runtime search to use its default search policy. The default search policy is sufficient for most scenarios.

Technical challenges

- It's not possible to load multiple versions of the runtime in a single process.

✖ Caution

Loading multiple copies or different versions of framework assemblies can lead to unexpected and hard-to-diagnose behavior.

💡 Tip

Use process boundaries with remoting or interprocess communication to solve this isolation problem.

- The timing of assembly loading can make testing and debugging difficult. Assemblies are typically loaded without their dependencies immediately being resolved. The dependencies are loaded as they are needed:
 - When code branches into a dependent assembly.
 - When code loads resources.
 - When code explicitly loads assemblies.
- The implementation of [AssemblyLoadContext.Load\(AssemblyName\)](#) can add new dependencies that may need to be isolated to allow different versions to exist. The

most natural implementation would place these dependencies in the default context. Careful design can isolate the new dependencies.

- The same assembly is loaded multiple times into different contexts.
 - This can lead to confusing error messages, for example "Unable to cast object of type 'Sample.Plugin' to type 'Sample.Plugin'".
 - Marshaling across isolation boundaries is non-trivial. A typical solution is to use an interface defined in an assembly that's only loaded into the default load context.

 **Collaborate with us on GitHub**

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

.NET is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

Common Language Runtime (CLR) overview

Article • 04/25/2023

.NET provides a run-time environment called the common language runtime that runs the code and provides services that make the development process easier.

Compilers and tools expose the common language runtime's functionality and enable you to write code that benefits from the managed execution environment. Code that you develop with a language compiler that targets the runtime is called managed code. Managed code benefits from features such as cross-language integration, cross-language exception handling, enhanced security, versioning and deployment support, a simplified model for component interaction, and debugging and profiling services.

ⓘ Note

Compilers and tools can produce output that the common language runtime can consume because the type system, the format of metadata, and the run-time environment (the virtual execution system) are all defined by a public standard, the ECMA Common Language Infrastructure specification. For more information, see [ECMA C# and Common Language Infrastructure Specifications](#).

To enable the runtime to provide services to managed code, language compilers must emit metadata that describes the types, members, and references in your code. Metadata is stored with the code; every loadable common language runtime portable executable (PE) file contains metadata. The runtime uses metadata to locate and load classes, lay out instances in memory, resolve method invocations, generate native code, enforce security, and set run-time context boundaries.

The runtime automatically handles object layout and manages references to objects, releasing them when they're no longer being used. Objects whose lifetimes are managed in this way are called managed data. Garbage collection eliminates memory leaks and some other common programming errors. If your code is managed, you can use managed, unmanaged, or both managed and unmanaged data in your .NET application. Because language compilers supply their own types, such as primitive types, you might not always know or need to know whether your data is being managed.

The common language runtime makes it easy to design components and applications whose objects interact across languages. Objects written in different languages can communicate with each other, and their behaviors can be tightly integrated. For

example, you can define a class and then use a different language to derive a class from your original class or call a method on the original class. You can also pass an instance of a class to a method of a class written in a different language. This cross-language integration is possible because language compilers and tools that target the runtime use a common type system defined by the runtime. They follow the runtime's rules for defining new types and for creating, using, persisting, and binding to types.

As part of their metadata, all managed components carry information about the components and resources they were built against. The runtime uses this information to ensure that your component or application has the specified versions of everything it needs, which makes your code less likely to break because of some unmet dependency. Registration information and state data are no longer stored in the registry, where they can be difficult to establish and maintain. Instead, information about the types you define and their dependencies is stored with the code as metadata. This way, the task of component replication and removal is less complicated.

Language compilers and tools expose the runtime's functionality in ways that are intended to be useful and intuitive to developers. Some features of the runtime might be more noticeable in one environment than in another. How you experience the runtime depends on which language compilers or tools you use. For example, if you're a Visual Basic developer, you might notice that with the common language runtime, the Visual Basic language has more object-oriented features than before. The runtime provides the following benefits:

- Performance improvements.
- The ability to easily use components developed in other languages.
- Extensible types provided by a class library.
- Language features such as inheritance, interfaces, and overloading for object-oriented programming.
- Support for explicit free threading that allows creation of multithreaded and scalable applications.
- Support for structured exception handling.
- Support for custom attributes.
- Garbage collection.
- Use of delegates instead of function pointers for increased type safety and security. For more information about delegates, see [Common Type System](#).

CLR versions

.NET Core and .NET 5+ releases have a single product version, that is, there's no separate CLR version. For a list of .NET Core versions, see [Download .NET Core](#).

However, the .NET Framework version number doesn't necessarily correspond to the version number of the CLR it includes. For a list of .NET Framework versions and their corresponding CLR versions, see [.NET Framework versions and dependencies](#).

Related articles

| Title | Description |
|---|---|
| Managed Execution Process | Describes the steps required to take advantage of the common language runtime. |
| Automatic Memory Management | Describes how the garbage collector allocates and releases memory. |
| Overview of .NET Framework | Describes key .NET Framework concepts, such as the common type system, cross-language interoperability, managed execution, application domains, and assemblies. |
| Common Type System | Describes how types are declared, used, and managed in the runtime in support of cross-language integration. |

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

Managed Execution Process

Article • 09/15/2021

The managed execution process includes the following steps, which are discussed in detail later in this topic:

1. Choosing a compiler.

To obtain the benefits provided by the common language runtime, you must use one or more language compilers that target the runtime.

2. Compiling your code to MSIL.

Compiling translates your source code into Microsoft intermediate language (MSIL) and generates the required metadata.

3. Compiling MSIL to native code.

At execution time, a just-in-time (JIT) compiler translates the MSIL into native code.

During this compilation, code must pass a verification process that examines the MSIL and metadata to find out whether the code can be determined to be type safe.

4. Running code.

The common language runtime provides the infrastructure that enables execution to take place and services that can be used during execution.

Choosing a Compiler

To obtain the benefits provided by the common language runtime (CLR), you must use one or more language compilers that target the runtime, such as Visual Basic, C#, Visual C++, F#, or one of many third-party compilers such as an Eiffel, Perl, or COBOL compiler.

Because it is a multilanguage execution environment, the runtime supports a wide variety of data types and language features. The language compiler you use determines which runtime features are available, and you design your code using those features.

Your compiler, not the runtime, establishes the syntax your code must use. If your component must be completely usable by components written in other languages, your component's exported types must expose only language features that are included in the Common Language Specification (CLS). You can use the [CLSCompliantAttribute](#) attribute to ensure that your code is CLS-compliant. For more information, see [Language independence and language-independent components](#).

[Back to top](#)

Compiling to MSIL

When compiling to managed code, the compiler translates your source code into Microsoft intermediate language (MSIL), which is a CPU-independent set of instructions that can be efficiently converted to native code. MSIL includes instructions for loading, storing, initializing, and calling methods on objects, as well as instructions for arithmetic and logical operations, control flow, direct memory access, exception handling, and other operations. Before code can be run, MSIL must be converted to CPU-specific code, usually by a [just-in-time \(JIT\) compiler](#). Because the common language runtime supplies one or more JIT compilers for each computer architecture it supports, the same set of MSIL can be JIT-compiled and run on any supported architecture.

When a compiler produces MSIL, it also produces metadata. Metadata describes the types in your code, including the definition of each type, the signatures of each type's members, the members that your code references, and other data that the runtime uses at execution time. The MSIL and metadata are contained in a portable executable (PE) file that is based on and that extends the published Microsoft PE and common object file format (COFF) used historically for executable content. This file format, which accommodates MSIL or native code as well as metadata, enables the operating system to recognize common language runtime images. The presence of metadata in the file together with MSIL enables your code to describe itself, which means that there is no need for type libraries or Interface Definition Language (IDL). The runtime locates and extracts the metadata from the file as needed during execution.

[Back to top](#)

Compiling MSIL to Native Code

Before you can run Microsoft intermediate language (MSIL), it must be compiled against the common language runtime to native code for the target machine architecture. .NET provides two ways to perform this conversion:

- A .NET just-in-time (JIT) compiler.
- [Ngen.exe \(Native Image Generator\)](#).

Compilation by the JIT Compiler

JIT compilation converts MSIL to native code on demand at application run time, when the contents of an assembly are loaded and executed. Because the common language runtime supplies a JIT compiler for each supported CPU architecture, developers can build a set of MSIL assemblies that can be JIT-compiled and run on different computers with different machine architectures. However, if your managed code calls platform-specific native APIs or a platform-specific class library, it will run only on that operating system.

JIT compilation takes into account the possibility that some code might never be called during execution. Instead of using time and memory to convert all the MSIL in a PE file to native code, it converts the MSIL as needed during execution and stores the resulting native code in memory so that it is accessible for subsequent calls in the context of that process. The loader creates and attaches a stub to each method in a type when the type is loaded and initialized. When a method is called for the first time, the stub passes control to the JIT compiler, which converts the MSIL for that method into native code and modifies the stub to point directly to the generated native code. Therefore, subsequent calls to the JIT-compiled method go directly to the native code.

Install-Time Code Generation Using NGen.exe

Because the JIT compiler converts an assembly's MSIL to native code when individual methods defined in that assembly are called, it affects performance adversely at run time. In most cases, that diminished performance is acceptable. More importantly, the code generated by the JIT compiler is bound to the process that triggered the compilation. It cannot be shared across multiple processes. To allow the generated code to be shared across multiple invocations of an application or across multiple processes that share a set of assemblies, the common language runtime supports an ahead-of-time compilation mode. This ahead-of-time compilation mode uses the [Ngen.exe \(Native Image Generator\)](#) to convert MSIL assemblies to native code much like the JIT compiler does. However, the operation of Ngen.exe differs from that of the JIT compiler in three ways:

- It performs the conversion from MSIL to native code before running the application instead of while the application is running.
- It compiles an entire assembly at a time, instead of one method at a time.
- It persists the generated code in the Native Image Cache as a file on disk.

Code Verification

As part of its compilation to native code, the MSIL code must pass a verification process unless an administrator has established a security policy that allows the code to bypass verification. Verification examines MSIL and metadata to find out whether the code is type safe, which means that it accesses only the memory locations it is authorized to access. Type safety helps isolate objects from each other and helps protect them from inadvertent or malicious corruption. It also provides assurance that security restrictions on code can be reliably enforced.

The runtime relies on the fact that the following statements are true for code that is verifiably type safe:

- A reference to a type is strictly compatible with the type being referenced.
- Only appropriately defined operations are invoked on an object.
- Identities are what they claim to be.

During the verification process, MSIL code is examined in an attempt to confirm that the code can access memory locations and call methods only through properly defined types. For example, code cannot allow an object's fields to be accessed in a manner that allows memory locations to be overrun. Additionally, verification inspects code to determine whether the MSIL has been correctly generated, because incorrect MSIL can lead to a violation of the type safety rules. The verification process passes a well-defined set of type-safe code, and it passes only code that is type safe. However, some type-safe code might not pass verification because of some limitations of the verification process, and some languages, by design, do not produce verifiably type-safe code. If type-safe code is required by the security policy but the code does not pass verification, an exception is thrown when the code is run.

[Back to top](#)

Running Code

The common language runtime provides the infrastructure that enables managed execution to take place and services that can be used during execution. Before a method can be run, it must be compiled to processor-specific code. Each method for which MSIL has been generated is JIT-compiled when it is called for the first time, and then run. The next time the method is run, the existing JIT-compiled native code is run. The process of JIT-compiling and then running the code is repeated until execution is complete.

During execution, managed code receives services such as garbage collection, security, interoperability with unmanaged code, cross-language debugging support, and

enhanced deployment and versioning support.

In Microsoft Windows Vista, the operating system loader checks for managed modules by examining a bit in the COFF header. The bit being set denotes a managed module. If the loader detects managed modules, it loads mscoree.dll, and `_CorValidateImage` and `_CorImageUnloading` notify the loader when the managed module images are loaded and unloaded. `_CorValidateImage` performs the following actions:

1. Ensures that the code is valid managed code.
2. Changes the entry point in the image to an entry point in the runtime.

On 64-bit Windows, `_CorValidateImage` modifies the image that is in memory by transforming it from PE32 to PE32+ format.

[Back to top](#)

See also

- [Overview](#)
- [Language independence and language-independent components](#)
- [Metadata and Self-Describing Components](#)
- [Ilasm.exe \(IL Assembler\)](#)
- [Security](#)
- [Interoperating with Unmanaged Code](#)
- [Deployment](#)
- [Assemblies in .NET](#)
- [Application Domains](#)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

Assemblies in .NET

Article • 03/15/2023

Assemblies are the fundamental units of deployment, version control, reuse, activation scoping, and security permissions for .NET-based applications. An assembly is a collection of types and resources that are built to work together and form a logical unit of functionality. Assemblies take the form of executable (.exe) or dynamic link library (.dll) files, and are the building blocks of .NET applications. They provide the common language runtime with the information it needs to be aware of type implementations.

In .NET and .NET Framework, you can build an assembly from one or more source code files. In .NET Framework, assemblies can contain one or more modules. This way, larger projects can be planned so that several developers can work on separate source code files or modules, which are combined to create a single assembly. For more information about modules, see [How to: Build a multifile assembly](#).

Assemblies have the following properties:

- Assemblies are implemented as *.exe* or *.dll* files.
- For libraries that target .NET Framework, you can share assemblies between applications by putting them in the [global assembly cache \(GAC\)](#). You must strong-name assemblies before you can include them in the GAC. For more information, see [Strong-named assemblies](#).
- Assemblies are only loaded into memory if they're required. If they aren't used, they aren't loaded. Therefore, assemblies can be an efficient way to manage resources in larger projects.
- You can programmatically obtain information about an assembly by using reflection. For more information, see [Reflection \(C#\)](#) or [Reflection \(Visual Basic\)](#).
- You can load an assembly just to inspect it by using the [MetadataLoadContext](#) class on .NET and .NET Framework. [MetadataLoadContext](#) replaces the [Assembly.ReflectionOnlyLoad](#) methods.

Assemblies in the common language runtime

Assemblies provide the common language runtime with the information it needs to be aware of type implementations. To the runtime, a type doesn't exist outside the context of an assembly.

An assembly defines the following information:

- **Code** that the common language runtime executes. Each assembly can have only one entry point: `DllMain`, `WinMain`, or `Main`.
- The **security boundary**. An assembly is the unit at which permissions are requested and granted. For more information about security boundaries in assemblies, see [Assembly security considerations](#).
- The **type boundary**. Every type's identity includes the name of the assembly in which it resides. A type called `MyType` that's loaded in the scope of one assembly isn't the same as a type called `MyType` that's loaded in the scope of another assembly.
- The **reference-scope boundary**: The [assembly manifest](#) has metadata that's used for resolving types and satisfying resource requests. The manifest specifies the types and resources to expose outside the assembly and enumerates other assemblies on which it depends. Microsoft intermediate language (MSIL) code in a portable executable (PE) file won't be executed unless it has an associated [assembly manifest](#).
- The **version boundary**. The assembly is the smallest versionable unit in the common language runtime. All types and resources in the same assembly are versioned as a unit. The [assembly manifest](#) describes the version dependencies you specify for any dependent assemblies. For more information about versioning, see [Assembly versioning](#).
- The **deployment unit**: When an application starts, only the assemblies that the application initially calls must be present. Other assemblies, such as assemblies containing localization resources or utility classes, can be retrieved on demand. This process allows apps to be simple and thin when first downloaded. For more information about deploying assemblies, see [Deploy applications](#).
- A **side-by-side execution unit**: For more information about running multiple versions of an assembly, see [Assemblies and side-by-side execution](#).

Create an assembly

Assemblies can be static or dynamic. Static assemblies are stored on a disk in portable executable (PE) files. Static assemblies can include interfaces, classes, and resources like bitmaps, JPEG files, and other resource files. You can also create dynamic assemblies, which are run directly from memory and aren't saved to disk before execution. You can save dynamic assemblies to disk after they've been executed.

There are several ways to create assemblies. You can use development tools, such as Visual Studio that can create `.dll` or `.exe` files. You can use tools in the Windows SDK to create assemblies with modules from other development environments. You can also use common language runtime APIs, such as [System.Reflection.Emit](#), to create dynamic assemblies.

Compile assemblies by building them in Visual Studio, building them with .NET Core command-line interface tools, or building .NET Framework assemblies with a command-line compiler. For more information about building assemblies using .NET CLI, see [.NET CLI overview](#).

 **Note**

To build an assembly in Visual Studio, on the **Build** menu, select **Build**.

Assembly manifest

Every assembly has an *assembly manifest* file. Similar to a table of contents, the assembly manifest contains:

- The assembly's identity (its name and version).
- A file table describing all the other files that make up the assembly, such as other assemblies you created that your `.exe` or `.dll` file relies on, bitmap files, or Readme files.
- An *assembly reference list*, which is a list of all external dependencies, such as `.dlls` or other files. Assembly references contain references to both global and private objects. Global objects are available to all other applications. In .NET Core, global objects are coupled with a particular .NET Core runtime. In .NET Framework, global objects reside in the global assembly cache (GAC). `System.IO.dll` is an example of an assembly in the GAC. Private objects must be in a directory level at or below the directory in which your app is installed.

Assemblies contain information about content, versioning, and dependencies. So the applications that use them don't need to rely on external sources, such as the registry on Windows systems, to function properly. Assemblies reduce `.dll` conflicts and make your applications more reliable and easier to deploy. In many cases, you can install a .NET-based application simply by copying its files to the target computer. For more information, see [Assembly manifest](#).

Add a reference to an assembly

To use an assembly in an application, you must add a reference to it. When an assembly is referenced, all the accessible types, properties, methods, and other members of its namespaces are available to your application as if their code were part of your source file.

ⓘ Note

Most assemblies from the .NET Class Library are referenced automatically. If a system assembly isn't automatically referenced, add a reference in one of the following ways:

- For .NET and .NET Core, add a reference to the NuGet package that contains the assembly. Either use the NuGet Package Manager in Visual Studio or add a `<PackageReference>` element for the assembly to the `.csproj` or `.vbproj` project.
- For .NET Framework, add a reference to the assembly by using the **Add Reference** dialog in Visual Studio or the `-reference` command line option for the **C#** or **Visual Basic** compilers.

In C#, you can use two versions of the same assembly in a single application. For more information, see [extern alias](#).

Related content

| Title | Description |
|--|---|
| Assembly contents | Elements that make up an assembly. |
| Assembly manifest | Data in the assembly manifest, and how it's stored in assemblies. |
| Global assembly cache | How the GAC stores and uses assemblies. |
| Strong-named assemblies | Characteristics of strong-named assemblies. |
| Assembly security considerations | How security works with assemblies. |
| Assembly versioning | Overview of the .NET Framework versioning policy. |
| Assembly placement | Where to locate assemblies. |

| Title | Description |
|---------------------------------------|---|
| Assemblies and side-by-side execution | Use multiple versions of the runtime or an assembly simultaneously. |
| Emit dynamic methods and assemblies | How to create dynamic assemblies. |
| How the runtime locates assemblies | How the .NET Framework resolves assembly references at run time. |

Reference

[System.Reflection.Assembly](#)

See also

- [.NET assembly file format](#)
- [Friend assemblies](#)
- [Reference assemblies](#)
- [How to: Load and unload assemblies](#)
- [How to: Use and debug assembly unloadability in .NET Core](#)
- [How to: Determine if a file is an assembly](#)
- [How to: Inspect assembly contents using MetadataLoadContext](#)

 Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

Assembly contents

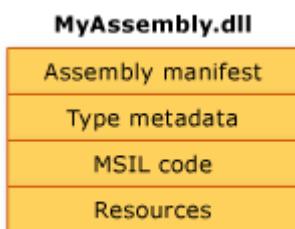
Article • 09/15/2021

In general, a static assembly can consist of four elements:

- The [assembly manifest](#), which contains assembly metadata.
- Type metadata.
- Microsoft intermediate language (MSIL) code that implements the types. It is generated by the compiler from one or more source code files.
- A set of [resources](#).

Only the assembly manifest is required, but either types or resources are needed to give the assembly any meaningful functionality.

The following illustration shows these elements grouped into a single physical file:



As you design your source code, you make explicit decisions about how to partition the functionality of your application into one or more files. When designing .NET code, you'll make similar decisions about how to partition the functionality into one or more assemblies.

See also

- [Assemblies in .NET](#)
- [Assembly manifest](#)
- [Assembly security considerations](#)

 Collaborate with us on
GitHub

The source for this content can
be found on GitHub, where you
can also create and review



.NET feedback

The .NET documentation is open
source. Provide feedback [here](#).

issues and pull requests. For more information, see [our contributor guide](#).

 [Open a documentation issue](#)

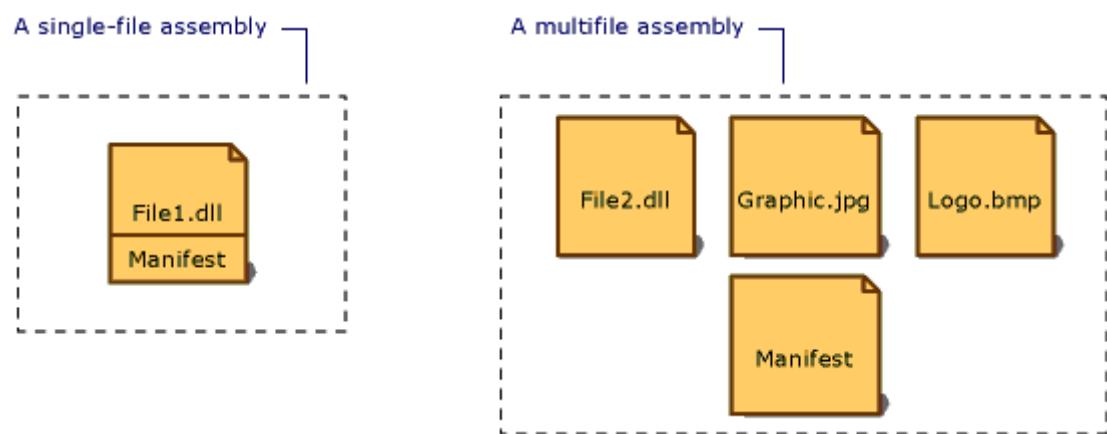
 [Provide product feedback](#)

Assembly manifest

Article • 09/15/2021

Every assembly, whether static or dynamic, contains a collection of data that describes how the elements in the assembly relate to each other. The assembly manifest contains this assembly metadata. An assembly manifest contains all the metadata needed to specify the assembly's version requirements and security identity, and all metadata needed to define the scope of the assembly and resolve references to resources and classes. The assembly manifest can be stored in either a PE file (an .exe or .dll) with Microsoft intermediate language (MSIL) code or in a standalone PE file that contains only assembly manifest information.

The following illustration shows the different ways the manifest can be stored.



For an assembly with one associated file, the manifest is incorporated into the PE file to form a single-file assembly. You can create a multifile assembly with a standalone manifest file or with the manifest incorporated into one of the PE files in the assembly.

Each assembly's manifest performs the following functions:

- Enumerates the files that make up the assembly.
- Governs how references to the assembly's types and resources map to the files that contain their declarations and implementations.
- Enumerates other assemblies on which the assembly depends.
- Provides a level of indirection between consumers of the assembly and the assembly's implementation details.
- Renders the assembly self-describing.

Assembly manifest contents

The following table shows the information contained in the assembly manifest. The first four items: assembly name, version number, culture, and strong name information make up the assembly's identity.

| Information | Description |
|--------------------------------------|---|
| Assembly name | A text string specifying the assembly's name. |
| Version number | A major and minor version number, and a revision and build number. The common language runtime uses these numbers to enforce version policy. |
| Culture | Information on the culture or language the assembly supports. This information should be used only to designate an assembly as a satellite assembly containing culture- or language-specific information. (An assembly with culture information is automatically assumed to be a satellite assembly.) |
| Strong name information | The public key from the publisher if the assembly has been given a strong name. |
| List of all files in the assembly | A hash of each file contained in the assembly and a file name. Note that all files that make up the assembly must be in the same directory as the file containing the assembly manifest. |
| Type reference information | Information used by the runtime to map a type reference to the file that contains its declaration and implementation. This is used for types that are exported from the assembly. |
| Information on referenced assemblies | A list of other assemblies that are statically referenced by the assembly. Each reference includes the dependent assembly's name, assembly metadata (version, culture, operating system, and so on), and public key, if the assembly is strong named. |

You can add or change some information in the assembly manifest by using assembly attributes in your code. You can change version information and informational attributes, including Trademark, Copyright, Product, Company, and Informational Version. For a complete list of assembly attributes, see [Set assembly attributes](#).

See also

- [Assembly contents](#)
- [Assembly versioning](#)
- [Create satellite assemblies](#)
- [Strong-named assemblies](#)

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

Assembly security considerations

Article • 03/30/2023

When you build an assembly, you can specify a set of permissions that the assembly requires to run. Whether certain permissions are granted or not granted to an assembly is based on evidence.

There are two distinct ways evidence is used:

- The input evidence is merged with the evidence gathered by the loader to create a final set of evidence used for policy resolution. The methods that use this semantic include `Assembly.Load`, `Assembly.LoadFrom`, and `Activator.CreateInstance`.
- The input evidence is used unaltered as the final set of evidence used for policy resolution. The methods that use this semantic include `Assembly.Load(byte[])` and `AppDomain.DefineDynamicAssembly()`.

Optional permissions can be granted by the [security policy](#) set on the computer where the assembly will run. If you want your code to handle all potential security exceptions, you can do one of the following:

- Insert a permission request for all the permissions your code must have, and handle up front the load-time failure that occurs if the permissions are not granted.
- Do not use a permission request to obtain permissions your code might need, but be prepared to handle security exceptions if permissions are not granted.

(!) Note

Security is a complex area, and you have many options to choose from. For more information, see [Key Security Concepts](#).

At load time, the assembly's evidence is used as input to security policy. Security policy is established by the enterprise and the computer's administrator as well as by user policy settings, and determines the set of permissions that's granted to all managed code when executed. Security policy can be established for the publisher of the assembly (if it has a signing tool generated signature), for the Web site and zone (which was an Internet Explorer concept) that the assembly was downloaded from, or for the assembly's strong name. For example, a computer's administrator can establish security policy that allows all code downloaded from a Web site and signed by a given software

company to access a database on a computer, but does not grant access to write to the computer's disk.

Strong-named assemblies and signing tools

⚠ Warning

Do not rely on strong names for security. They provide a unique identity only.

You can sign an assembly in two different but complementary ways: with a strong name or by using [SignTool.exe \(Sign Tool\)](#). Signing an assembly with a strong name adds public key encryption to the file containing the assembly manifest. Strong name signing helps to verify name uniqueness, prevent name spoofing, and provide callers with some identity when a reference is resolved.

No level of trust is associated with a strong name, which makes [SignTool.exe \(Sign Tool\)](#) important. The two signing tools require a publisher to prove its identity to a third-party authority and obtain a certificate. This certificate is then embedded in your file and can be used by an administrator to decide whether to trust the code's authenticity.

You can give both a strong name and a digital signature created using [SignTool.exe \(Sign Tool\)](#) to an assembly, or you can use either alone. The two signing tools can sign only one file at a time; for a multifile assembly, you sign the file that contains the assembly manifest. A strong name is stored in the file containing the assembly manifest, but a signature created using [SignTool.exe \(Sign Tool\)](#) is stored in a reserved slot in the portable executable (PE) file containing the assembly manifest. Signing of an assembly using [SignTool.exe \(Sign Tool\)](#) can be used (with or without a strong name) when you already have a trust hierarchy that relies on [SignTool.exe \(Sign Tool\)](#) generated signatures, or when your policy uses only the key portion and does not check a chain of trust.

ⓘ Note

When using both a strong name and a signing tool signature on an assembly, the strong name must be assigned first.

The common language runtime also performs a hash verification; the assembly manifest contains a list of all files that make up the assembly, including a hash of each file as it existed when the manifest was built. As each file is loaded, its contents are hashed and

compared with the hash value stored in the manifest. If the two hashes do not match, the assembly fails to load.

Strong naming and signing using [SignTool.exe \(Sign Tool\)](#) guarantee integrity through digital signatures and certificates. All the technologies mentioned, that is, hash verification, strong naming, and signing using [SignTool.exe \(Sign Tool\)](#), work together to ensure that the assembly has not been altered in any way.

See also

- [Strong-named assemblies](#)
- [Assemblies in .NET](#)
- [SignTool.exe \(Sign Tool\)](#)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

Assembly versioning

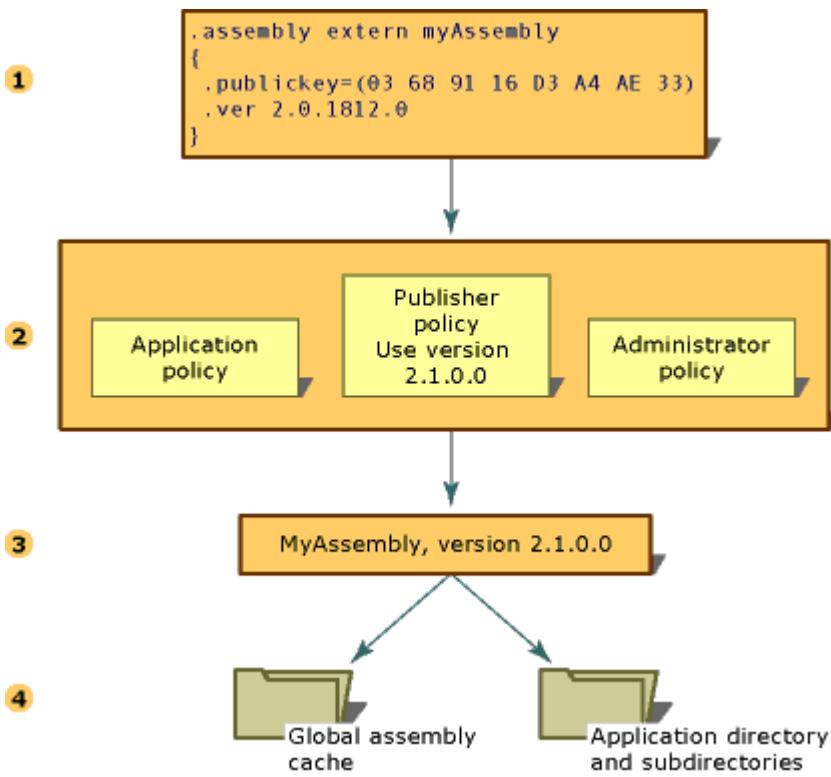
Article • 10/01/2021

All versioning of assemblies that use the common language runtime is done at the assembly level. The specific version of an assembly and the versions of dependent assemblies are recorded in the assembly's manifest. The default version policy for the runtime is that applications run only with the versions they were built and tested with, unless overridden by explicit version policy in configuration files (the application configuration file, the publisher policy file, and the computer's administrator configuration file).

The runtime performs several steps to resolve an assembly binding request:

1. Checks the original assembly reference to determine the version of the assembly to be bound.
2. Checks for all applicable configuration files to apply version policy.
3. Determines the correct assembly from the original assembly reference and any redirection specified in the configuration files, and determines the version that should be bound to the calling assembly.
4. Checks the global assembly cache, codebases specified in configuration files, and then checks the application's directory and subdirectories using the probing rules explained in [How the runtime locates assemblies](#).

The following illustration shows these steps:



For more information about configuring applications, see [Configure apps](#). For more information about binding policy, see [How the runtime locates assemblies](#).

Version information

Each assembly has two distinct ways of expressing version information:

- The assembly's version number, which, together with the assembly name and culture information, is part of the assembly's identity. This number is used by the runtime to enforce version policy and plays a key part in the type resolution process at run time.
- An informational version, which is a string that represents additional version information included for informational purposes only.

Assembly version number

Each assembly has a version number as part of its identity. As such, two assemblies that differ by version number are considered by the runtime to be completely different assemblies. This version number is physically represented as a four-part string with the following format:

<major version>. <minor version>. <build number>. <revision>

For example, version 1.5.1254.0 indicates 1 as the major version, 5 as the minor version, 1254 as the build number, and 0 as the revision number.

The version number is stored in the assembly manifest along with other identity information, including the assembly name and public key, as well as information on relationships and identities of other assemblies connected with the application.

When an assembly is built, the development tool records dependency information for each assembly that is referenced in the assembly manifest. The runtime uses these version numbers, in conjunction with configuration information set by an administrator, an application, or a publisher, to load the proper version of a referenced assembly.

The runtime distinguishes between regular and strong-named assemblies for the purposes of versioning. Version checking only occurs with strong-named assemblies.

For information about specifying version binding policies, see [Configure apps](#). For information about how the runtime uses version information to find a particular assembly, see [How the runtime locates assemblies](#).

Assembly informational version

The informational version is a string that attaches additional version information to an assembly for informational purposes only; this information is not used at run time. The text-based informational version corresponds to the product's marketing literature, packaging, or product name and is not used by the runtime. For example, an informational version could be "Common Language Runtime version 1.0" or "NET Control SP 2". On the Version tab of the file properties dialog in Microsoft Windows, this information appears in the item "Product Version".

ⓘ Note

Although you can specify any text, a warning message appears on compilation if the string is not in the format used by the assembly version number, or if it is in that format but contains wildcards. This warning is harmless.

The informational version is represented using the custom attribute [System.Reflection.AssemblyInformationalVersionAttribute](#). For more information about the informational version attribute, see [Set assembly attributes](#).

See also

- [How the runtime locates assemblies](#)
- [Configure apps](#)
- [Set assembly attributes](#)

- Assemblies in .NET

 **Collaborate with us on GitHub**

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

 **.NET feedback**

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

System.Version class

Article • 01/08/2024

This article provides supplementary remarks to the reference documentation for this API.

The [Version class](#) represents the version number of an assembly, operating system, or the common language runtime. Version numbers consist of two to four components: major, minor, build, and revision. The major and minor components are required; the build and revision components are optional, but the build component is required if the revision component is defined. All defined components must be integers greater than or equal to 0. The format of the version number is as follows (optional components are shown in square brackets):

major.minor[.build[.revision]]

The components are used by convention as follows:

- *Major*: Assemblies with the same name but different major versions are not interchangeable. A higher version number might indicate a major rewrite of a product where backward compatibility cannot be assumed.
- *Minor*: If the name and major version number on two assemblies are the same, but the minor version number is different, this indicates significant enhancement with the intention of backward compatibility. This higher minor version number might indicate a point release of a product or a fully backward-compatible new version of a product.
- *Build*: A difference in build number represents a recompilation of the same source. Different build numbers might be used when the processor, platform, or compiler changes.
- *Revision*: Assemblies with the same name, major, and minor version numbers but different revisions are intended to be fully interchangeable. A higher revision number might be used in a build that fixes a security hole in a previously released assembly.

Subsequent versions of an assembly that differ only by build or revision numbers are considered to be Hotfix updates of the prior version.

 **Important**

The value of [Version](#) properties that have not been explicitly assigned a value is undefined (-1).

The [MajorRevision](#) and [MinorRevision](#) properties enable you to identify a temporary version of your application that, for example, corrects a problem until you can release a permanent solution. Furthermore, the Windows NT operating system uses the [MajorRevision](#) property to encode the service pack number.

Assign version information to assemblies

Ordinarily, the [Version](#) class is not used to assign a version number to an assembly. Instead, the [AssemblyVersionAttribute](#) class is used to define an assembly's version, as illustrated by the example in this article.

Retrieve version information

[Version](#) objects are most frequently used to store version information about some system or application component (such as the operating system), the common language runtime, the current application's executable, or a particular assembly. The following examples illustrate some of the most common scenarios:

- Retrieving the operating system version. The following example uses the [OperatingSystem.Version](#) property to retrieve the version number of the operating system.

C#

```
// Get the operating system version.  
OperatingSystem os = Environment.OSVersion;  
Version ver = os.Version;  
Console.WriteLine("Operating System: {0} ({1})", os.VersionString,  
ver.ToString());
```

- Retrieving the version of the common language runtime. The following example uses the [Environment.Version](#) property to retrieve version information about the common language runtime.

C#

```
// Get the common language runtime version.  
Version ver = Environment.Version;  
Console.WriteLine("CLR Version {0}", ver.ToString());
```

- Retrieving the current application's assembly version. The following example uses the [Assembly.GetEntryAssembly](#) method to obtain a reference to an [Assembly](#) object that represents the application executable and then retrieves its assembly version number.

C#

```
using System;
using System.Reflection;

public class Example4
{
    public static void Main()
    {
        // Get the version of the executing assembly (that is, this
        // assembly).
        Assembly assem = Assembly.GetEntryAssembly();
        AssemblyName assemName = assem.GetName();
        Version ver = assemName.Version;
        Console.WriteLine("Application {0}, Version {1}", assemName.Name,
        ver.ToString());
    }
}
```

- Retrieving the current assembly's assembly version. The following example uses the [Type.Assembly](#) property to obtain a reference to an [Assembly](#) object that represents the assembly that contains the application entry point, and then retrieves its version information.

C#

```
using System;
using System.Reflection;

public class Example3
{
    public static void Main()
    {
        // Get the version of the current assembly.
        Assembly assem = typeof(Example).Assembly;
        AssemblyName assemName = assem.GetName();
        Version ver = assemName.Version;
        Console.WriteLine("{0}, Version {1}", assemName.Name,
        ver.ToString());
    }
}
```

- Retrieving the version of a specific assembly. The following example uses the [Assembly.ReflectionOnlyLoadFrom](#) method to obtain a reference to an [Assembly](#)

object that has a particular file name, and then retrieves its version information. Note that several other methods also exist to instantiate an [Assembly](#) object by file name or by strong name.

C#

```
using System;
using System.Reflection;

public class Example5
{
    public static void Main()
    {
        // Get the version of a specific assembly.
        string filename = @".\StringLibrary.dll";
        Assembly assem = Assembly.ReflectionOnlyLoadFrom(filename);
        AssemblyName assemName = assem.GetName();
        Version ver = assemName.Version;
        Console.WriteLine("{0}, Version {1}", assemName.Name,
ver.ToString());
    }
}
```

- Retrieving the Publish Version of a ClickOnce application. The following example uses the [ApplicationDeployment.CurrentVersion](#) property to display an application's Publish Version. Note that its successful execution requires the example's application identity to be set. This is handled automatically by the Visual Studio Publish Wizard.

C#

```
using System;
using System.Deployment.Application;

public class Example
{
    public static void Main()
    {
        Version ver =
ApplicationDeployment.CurrentDeployment.CurrentVersion;
        Console.WriteLine("ClickOnce Publish Version: {0}", ver);
    }
}
```

 **Important**

The Publish Version of an application for ClickOnce deployment is completely independent of its assembly version.

Compare version objects

You can use the [CompareTo](#) method to determine whether one [Version](#) object is earlier than, the same as, or later than a second [Version](#) object. The following example indicates that Version 2.1 is later than Version 2.0.

C#

```
Version v1 = new Version(2, 0);
Version v2 = new Version("2.1");
Console.WriteLine("Version {0} is ", v1);
switch(v1.CompareTo(v2))
{
    case 0:
        Console.WriteLine("the same as");
        break;
    case 1:
        Console.WriteLine("later than");
        break;
    case -1:
        Console.WriteLine("earlier than");
        break;
}
Console.WriteLine(" Version {0}.", v2);
// The example displays the following output:
//      Version 2.0 is earlier than Version 2.1.
```

For two versions to be equal, the major, minor, build, and revision numbers of the first [Version](#) object must be identical to those of the second [Version](#) object. If the build or revision number of a [Version](#) object is undefined, that [Version](#) object is considered to be earlier than a [Version](#) object whose build or revision number is equal to zero. The following example illustrates this by comparing three [Version](#) objects that have undefined version components.

C#

```
using System;

enum VersionTime {Earlier = -1, Same = 0, Later = 1};

public class Example2
{
    public static void Main()
    {
```

```
Version v1 = new Version(1, 1);
Version v1a = new Version("1.1.0");
ShowRelationship(v1, v1a);

Version v1b = new Version(1, 1, 0, 0);
ShowRelationship(v1b, v1a);
}

private static void ShowRelationship(Version v1, Version v2)
{
    Console.WriteLine("Relationship of {0} to {1}: {2}",
                      v1, v2, (VersionTime) v1.CompareTo(v2));
}
}

// The example displays the following output:
//      Relationship of 1.1 to 1.1.0: Earlier
//      Relationship of 1.1.0.0 to 1.1.0: Later
```

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



.NET feedback

.NET is an open source project.
Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

Assemblies and side-by-side execution

Article • 09/15/2021

Side-by-side execution is the ability to store and execute multiple versions of an application or component on the same computer. This means that you can have multiple versions of the runtime, and multiple versions of applications and components that use a version of the runtime, on the same computer at the same time. Side-by-side execution gives you more control over what versions of a component an application binds to, and more control over what version of the runtime an application uses.

Support for side-by-side storage and execution of different versions of the same assembly is an integral part of strong naming and is built into the infrastructure of the runtime. Because the strong-named assembly's version number is part of its identity, the runtime can store multiple versions of the same assembly in the global assembly cache and load those assemblies at run time.

Although the runtime provides you with the ability to create side-by-side applications, side-by-side execution is not automatic. For more information on creating applications for side-by-side execution, see [Guidelines for creating components for side-by-side execution](#).

See also

- [How the runtime locates assemblies](#)
- [Assemblies in .NET](#)

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

.NET assembly file format

Article • 09/15/2021

.NET defines a binary file format, *assembly*, that is used to fully describe and contain .NET programs. Assemblies are used for the programs themselves as well as any dependent libraries. A .NET program can be executed as one or more assemblies, with no other required artifacts, beyond the appropriate .NET implementation. Native dependencies, including operating system APIs, are a separate concern and are not contained within the .NET assembly format, although they are sometimes described with this format (for example, WinRT).

Each CLI component carries the metadata for declarations, implementations, and references specific to that component. Therefore, the component-specific metadata is referred to as component metadata, and the resulting component is said to be self-describing – from ECMA 335 I.9.1, Components and assemblies.

The format is fully specified and standardized as [ECMA 335](#). All .NET compilers and runtimes use this format. The presence of a documented and infrequently updated binary format has been a major benefit (arguably a requirement) for interoperability. The format was last updated in a substantive way in 2005 (.NET Framework 2.0) to accommodate generics and processor architecture.

The format is CPU- and OS-agnostic. It has been used as part of .NET implementations that target many chips and CPUs. While the format itself has Windows heritage, it is implementable on any operating system. Its arguably most significant choice for OS interoperability is that most values are stored in little-endian format. It doesn't have a specific affinity to machine pointer size (for example, 32-bit, 64-bit).

The .NET assembly format is also very descriptive about the structure of a given program or library. It describes the internal components of an assembly, specifically assembly references and types defined and their internal structure. Tools or APIs can read and process this information for display or to make programmatic decisions.

Format

The .NET binary format is based on the Windows [PE file](#) format. In fact, .NET class libraries are conformant Windows PEs, and appear on first glance to be Windows dynamic link libraries (DLLs) or application executables (EXEs). This is a very useful characteristic on Windows, where they can masquerade as native executable binaries and get some of the same treatment (for example, OS load, PE tools).

PE Headers

CLI Header

CLI Data : metadata, IL method bodies, fix-ups

Native Image Sections

Assembly Headers from ECMA 335 II.25.1, Structure of the runtime file format.

Process the assemblies

It is possible to write tools or APIs to process assemblies. Assembly information enables making programmatic decisions at run time, re-writing assemblies, providing API IntelliSense in an editor and generating documentation. [System.Reflection](#), [System.Reflection.MetadataLoadContext](#), and [Mono.Cecil](#) ↗ are good examples of tools that are frequently used for this purpose.

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

How to use and debug assembly unloadability in .NET

Article • 11/13/2023

.NET (Core) introduced the ability to load and later unload a set of assemblies. In .NET Framework, custom app domains were used for this purpose, but .NET (Core) only supports a single default app domain.

Unloadability is supported through [AssemblyLoadContext](#). You can load a set of assemblies into a collectible `AssemblyLoadContext`, execute methods in them or just inspect them using reflection, and finally unload the `AssemblyLoadContext`. That unloads the assemblies loaded into the `AssemblyLoadContext`.

There's one noteworthy difference between the unloading using `AssemblyLoadContext` and using AppDomains. With AppDomains, the unloading is forced. At unload time, all threads running in the target AppDomain are aborted, managed COM objects created in the target AppDomain are destroyed, and so on. With `AssemblyLoadContext`, the unload is "cooperative". Calling the [AssemblyLoadContext.Unload](#) method just initiates the unloading. The unloading finishes after:

- No threads have methods from the assemblies loaded into the `AssemblyLoadContext` on their call stacks.
- None of the types from the assemblies loaded into the `AssemblyLoadContext`, instances of those types, and the assemblies themselves are referenced by:
 - References outside of the `AssemblyLoadContext`, except for weak references ([WeakReference](#) or [WeakReference<T>](#)).
 - Strong garbage collector (GC) handles ([GCHandleType.Normal](#) or [GCHandleType.Pinned](#)) from both inside and outside of the `AssemblyLoadContext`.

Use collectible `AssemblyLoadContext`

This section contains a detailed step-by-step tutorial that shows a simple way to load a .NET (Core) application into a collectible `AssemblyLoadContext`, execute its entry point, and then unload it. You can find a complete sample at [https://github.com/dotnet/samples/tree/main/core/tutorials/Unloading ↗](https://github.com/dotnet/samples/tree/main/core/tutorials/Unloading).

Create a collectible `AssemblyLoadContext`

Derive your class from the [AssemblyLoadContext](#) and override its [AssemblyLoadContext.Load](#) method. That method resolves references to all assemblies that are dependencies of assemblies loaded into that [AssemblyLoadContext](#).

The following code is an example of the simplest custom [AssemblyLoadContext](#):

```
C#  
  
class TestAssemblyLoadContext : AssemblyLoadContext  
{  
    public TestAssemblyLoadContext() : base(isCollectible: true)  
    {  
    }  
  
    protected override Assembly? Load(AssemblyName name)  
    {  
        return null;  
    }  
}
```

As you can see, the `Load` method returns `null`. That means that all the dependency assemblies are loaded into the default context, and the new context contains only the assemblies explicitly loaded into it.

If you want to load some or all of the dependencies into the [AssemblyLoadContext](#) too, you can use the [AssemblyDependencyResolver](#) in the `Load` method. The [AssemblyDependencyResolver](#) resolves the assembly names to absolute assembly file paths. The resolver uses the `.deps.json` file and assembly files in the directory of the main assembly loaded into the context.

```
C#  
  
using System.Reflection;  
using System.Runtime.Loader;  
  
namespace complex  
{  
    class TestAssemblyLoadContext : AssemblyLoadContext  
    {  
        private AssemblyDependencyResolver _resolver;  
  
        public TestAssemblyLoadContext(string mainAssemblyToLoadPath) :  
base(isCollectible: true)  
        {  
            _resolver = new  
AssemblyDependencyResolver(mainAssemblyToLoadPath);  
        }  
  
        protected override Assembly? Load(AssemblyName name)
```

```
        {
            string? assemblyPath = _resolver.ResolveAssemblyToPath(name);
            if (assemblyPath != null)
            {
                return LoadFromAssemblyPath(assemblyPath);
            }

            return null;
        }
    }
}
```

Use a custom collectible AssemblyLoadContext

This section assumes the simpler version of the `TestAssemblyLoadContext` is being used.

You can create an instance of the custom `AssemblyLoadContext` and load an assembly into it as follows:

C#

```
var alc = new TestAssemblyLoadContext();
Assembly a = alc.LoadFromAssemblyPath(assemblyPath);
```

For each of the assemblies referenced by the loaded assembly, the `TestAssemblyLoadContext.Load` method is called so that the `TestAssemblyLoadContext` can decide where to get the assembly from. In this case, it returns `null` to indicate that it should be loaded into the default context from locations that the runtime uses to load assemblies by default.

Now that an assembly was loaded, you can execute a method from it. Run the `Main` method:

C#

```
var args = new object[1] {new string[] {"Hello"}};
_ = a.EntryPoint?.Invoke(null, args);
```

After the `Main` method returns, you can initiate unloading by either calling the `Unload` method on the custom `AssemblyLoadContext` or removing the reference you have to the `AssemblyLoadContext`:

C#

```
alc.Unload();
```

This is sufficient to unload the test assembly. Next, you'll put all of this into a separate noninlineable method to ensure that the `TestAssemblyLoadContext`, `Assembly`, and `MethodInfo` (the `Assembly.EntryPoint`) can't be kept alive by stack slot references (real- or JIT-introduced locals). That could keep the `TestAssemblyLoadContext` alive and prevent the unload.

Also, return a weak reference to the `AssemblyLoadContext` so that you can use it later to detect unload completion.

C#

```
[MethodImpl(MethodImplOptions.NoInlining)]
static void ExecuteAndUnload(string assemblyPath, out WeakReference
alcWeakRef)
{
    var alc = new TestAssemblyLoadContext();
    Assembly a = alc.LoadFromAssemblyPath(assemblyPath);

    alcWeakRef = new WeakReference(alc, trackResurrection: true);

    var args = new object[1] {new string[] {"Hello"}};
    _ = a.EntryPoint?.Invoke(null, args);

    alc.Unload();
}
```

Now you can run this function to load, execute, and unload the assembly.

C#

```
WeakReference testAlcWeakRef;
ExecuteAndUnload("absolute/path/to/your/assembly", out testAlcWeakRef);
```

However, the unload doesn't complete immediately. As previously mentioned, it relies on the garbage collector to collect all the objects from the test assembly. In many cases, it isn't necessary to wait for the unload completion. However, there are cases where it's useful to know that the unload has finished. For example, you might want to delete the assembly file that was loaded into the custom `AssemblyLoadContext` from disk. In such a case, the following code snippet can be used. It triggers garbage collection and waits for pending finalizers in a loop until the weak reference to the custom `AssemblyLoadContext` is set to `null`, indicating the target object was collected. In most cases, just one pass through the loop is required. However, for more complex cases where objects created

by the code running in the `AssemblyLoadContext` have finalizers, more passes might be needed.

C#

```
for (int i = 0; testAlcWeakRef.IsAlive && (i < 10); i++)
{
    GC.Collect();
    GC.WaitForPendingFinalizers();
}
```

The Unloading event

In some cases, it might be necessary for the code loaded into a custom `AssemblyLoadContext` to perform some cleanup when the unloading is initiated. For example, it might need to stop threads or clean up strong GC handles. The `Unloading` event can be used in such cases. You can hook a handler that performs the necessary cleanup to this event.

Troubleshoot unloadability issues

Due to the cooperative nature of the unloading, it's easy to forget about references that might be keeping the stuff in a collectible `AssemblyLoadContext` alive and preventing unload. Here's a summary of entities (some of them nonobvious) that can hold the references:

- Regular references held from outside of the collectible `AssemblyLoadContext` that are stored in a stack slot or a processor register (method locals, either explicitly created by the user code or implicitly by the just-in-time (JIT) compiler), a static variable, or a strong (pinning) GC handle, and transitively pointing to:
 - An assembly loaded into the collectible `AssemblyLoadContext`.
 - A type from such an assembly.
 - An instance of a type from such an assembly.
- Threads running code from an assembly loaded into the collectible `AssemblyLoadContext`.
- Instances of custom, noncollectible `AssemblyLoadContext` types created inside of the collectible `AssemblyLoadContext`.
- Pending `RegisteredWaitHandle` instances with callbacks set to methods in the custom `AssemblyLoadContext`.

Tip

Object references that are stored in stack slots or processor registers and that could prevent unloading of an `AssemblyLoadContext` can occur in the following situations:

- When function call results are passed directly to another function, even though there is no user-created local variable.
- When the JIT compiler keeps a reference to an object that was available at some point in a method.

Debug unloading issues

Debugging issues with unloading can be tedious. You can get into situations where you don't know what can be holding an `AssemblyLoadContext` alive, but the unload fails. The best tool to help with that is WinDbg (or LLDB on Unix) with the SOS plugin. You need to find what's keeping a `LoaderAllocator` that belongs to the specific `AssemblyLoadContext` alive. The SOS plugin lets you look at GC heap objects, their hierarchies, and roots.

To load the SOS plugin into the debugger, enter one of the following commands in the debugger command line.

In WinDbg (if it's not already loaded):

```
Console  
.loadby sos coreclr
```

In LLDB:

```
Console  
plugin load /path/to/libssosplugin.so
```

Now you'll debug an example program that has problems with unloading. The source code is available in the [Example source code](#) section. When you run it under WinDbg, the program breaks into the debugger right after attempting to check for the unload success. You can then start looking for the culprits.

 **Tip**

If you debug using LLDB on Unix, the SOS commands in the following examples don't have the ! in front of them.

Console

```
!dumpheap -type LoaderAllocator
```

This command dumps all objects with a type name containing `LoaderAllocator` that are in the GC heap. Here's an example:

Console

| Address | MT | Size |
|------------------|------------------|------|
| 000002b78000ce40 | 00007ffadc93a288 | 48 |
| 000002b78000ceb0 | 00007ffadc93a218 | 24 |

Statistics:

| MT | Count | TotalSize | Class | Name |
|--|-------|-----------|-----------------------------------|------|
| 00007ffadc93a218 | 1 | 24 | | |
| System.Reflection.LoaderAllocatorScout | | | | |
| 00007ffadc93a288 | 1 | 48 | System.Reflection.LoaderAllocator | |
| Total | 2 | objects | | |

In the "Statistics:" part, check the `MT` (`MethodTable`) that belongs to the `System.Reflection.LoaderAllocator`, which is the object you care about. Then, in the list at the beginning, find the entry with `MT` that matches that one, and get the address of the object itself. In this case, it's "000002b78000ce40".

Now that you know the address of the `LoaderAllocator` object, you can use another command to find its GC roots:

Console

```
!gcroot 0x000002b78000ce40
```

This command dumps the chain of object references that lead to the `LoaderAllocator` instance. The list starts with the root, which is the entity that keeps the `LoaderAllocator` alive and thus is the core of the problem. The root can be a stack slot, a processor register, a GC handle, or a static variable.

Here's an example of the output of the `gcroot` command:

Console

```

Thread 4ac:
  000000cf9499dd20 00007ffa7d0236bc example.Program.Main(System.String[])
[E:\unloadability\example\Program.cs @ 70]
  rbp-20: 000000cf9499dd90
    -> 000002b78000d328 System.Reflection.RuntimeMethodInfo
    -> 000002b78000d1f8 System.RuntimeType+RuntimeTypeCache
    -> 000002b78000d1d0 System.RuntimeType
    -> 000002b78000ce40 System.Reflection.LoaderAllocator

HandleTable:
  000002b7f8a81198 (strong handle)
  -> 000002b78000d948 test.Test
  -> 000002b78000ce40 System.Reflection.LoaderAllocator

  000002b7f8a815f8 (pinned handle)
  -> 000002b790001038 System.Object[]
  -> 000002b78000d390 example.TestInfo
  -> 000002b78000d328 System.Reflection.RuntimeMethodInfo
  -> 000002b78000d1f8 System.RuntimeType+RuntimeTypeCache
  -> 000002b78000d1d0 System.RuntimeType
  -> 000002b78000ce40 System.Reflection.LoaderAllocator

Found 3 roots.

```

The next step is to figure out where the root is located so you can fix it. The easiest case is when the root is a stack slot or a processor register. In that case, the `gcroot` shows the name of the function whose frame contains the root and the thread executing that function. The difficult case is when the root is a static variable or a GC handle.

In the previous example, the first root is a local of type `System.Reflection.RuntimeMethodInfo` stored in the frame of the function `example.Program.Main(System.String[])` at address `rbp-20` (`rbp` is the processor register `rbp` and `-20` is a hexadecimal offset from that register).

The second root is a normal (strong) `GCHandle` that holds a reference to an instance of the `test.Test` class.

The third root is a pinned `GCHandle`. This one is actually a static variable, but unfortunately, there's no way to tell. Statics for reference types are stored in a managed object array in internal runtime structures.

Another case that can prevent unloading of an `AssemblyLoadContext` is when a thread has a frame of a method from an assembly loaded into the `AssemblyLoadContext` on its stack. You can check that by dumping managed call stacks of all threads:

Console

```
~*e !clrstack
```

The command means "apply to all threads the `!clrstack` command". The following is the output of that command for the example. Unfortunately, LLDB on Unix doesn't have any way to apply a command to all threads, so you must manually switch threads and repeat the `clrstack` command. Ignore all threads where the debugger says "Unable to walk the managed stack".

Console

```
OS Thread Id: 0x6ba8 (0)
    Child SP          IP Call Site
0000001fc697d5c8 00007ffb50d9de12 [HelperMethodFrame: 0000001fc697d5c8]
System.Diagnostics.Debugger.BreakInternal()
0000001fc697d6d0 00007ffa864765fa System.Diagnostics.Debugger.Break()
0000001fc697d700 00007ffa864736bc example.Program.Main(System.String[])
[E:\unloadability\example\Program.cs @ 70]
0000001fc697d998 00007ffae5fdc1e3 [GCFrame: 0000001fc697d998]
0000001fc697df28 00007ffae5fdc1e3 [GCFrame: 0000001fc697df28]
OS Thread Id: 0x2ae4 (1)
Unable to walk the managed stack. The current thread is likely not a
managed thread. You can run !threads to get a list of managed threads in
the process
Failed to start stack walk: 80070057
OS Thread Id: 0x61a4 (2)
Unable to walk the managed stack. The current thread is likely not a
managed thread. You can run !threads to get a list of managed threads in
the process
Failed to start stack walk: 80070057
OS Thread Id: 0x7fdc (3)
Unable to walk the managed stack. The current thread is likely not a
managed thread. You can run !threads to get a list of managed threads in
the process
Failed to start stack walk: 80070057
OS Thread Id: 0x5390 (4)
Unable to walk the managed stack. The current thread is likely not a
managed thread. You can run !threads to get a list of managed threads in
the process
Failed to start stack walk: 80070057
OS Thread Id: 0x5ec8 (5)
    Child SP          IP Call Site
0000001fc70ff6e0 00007ffb5437f6e4 [DebuggerU2MCatchHandlerFrame:
0000001fc70ff6e0]
OS Thread Id: 0x4624 (6)
    Child SP          IP Call Site
GetFrameContext failed: 1
0000000000000000 0000000000000000
OS Thread Id: 0x60bc (7)
    Child SP          IP Call Site
0000001fc727f158 00007ffb5437fce4 [HelperMethodFrame: 0000001fc727f158]
System.Threading.Thread.SleepInternal(Int32)
```

```
0000001fc727f260 00007ffb37ea7c2b System.Threading.Thread.Sleep(Int32)
0000001fc727f290 00007ffa865005b3 test.Program.ThreadProc()
[E:\unloadability\test\Program.cs @ 17]
0000001fc727f2c0 00007ffb37ea6a5b
System.Threading.Thread.ThreadMain_ThreadStart()
0000001fc727f2f0 00007ffadbc4cbe3
System.Threading.ExecutionContext.RunInternal(System.Threading.ExecutionCont
ext, System.Threading.ContextCallback, System.Object)
0000001fc727f568 00007ffae5fdc1e3 [GCFrame: 0000001fc727f568]
0000001fc727f7f0 00007ffae5fdc1e3 [DebuggerU2MCatchHandlerFrame:
0000001fc727f7f0]
```

As you can see, the last thread has `test.Program.ThreadProc()`. This is a function from the assembly loaded into the `AssemblyLoadContext`, and so it keeps the `AssemblyLoadContext` alive.

Example source code

The following code that contains unloadability issues is used in the previous debugging example.

Main testing program

```
C#  
  
using System;
using System.Reflection;
using System.Runtime.CompilerServices;
using System.Runtime.Loader;  
  
namespace example
{
    class TestAssemblyLoadContext : AssemblyLoadContext
    {
        public TestAssemblyLoadContext() : base(true)
        {
        }
        protected override Assembly? Load(AssemblyName name)
        {
            return null;
        }
    }

    class TestInfo
    {
        public TestInfo(MethodInfo? mi)
        {
            _entryPoint = mi;
        }
    }
}
```

```
        MethodInfo? _entryPoint;
    }

    class Program
    {
        static TestInfo? entryPoint;

        [MethodImpl(MethodImplOptions.NoInlining)]
        static int ExecuteAndUnload(string assemblyPath, out WeakReference testAlcWeakRef, out MethodInfo? testEntryPoint)
        {
            var alc = new TestAssemblyLoadContext();
            testAlcWeakRef = new WeakReference(alc);

            Assembly a = alc.LoadFromAssemblyPath(assemblyPath);
            if (a == null)
            {
                testEntryPoint = null;
                Console.WriteLine("Loading the test assembly failed");
                return -1;
            }

            var args = new object[1] {new string[] {"Hello"}};

            // Issue preventing unloading #1 - we keep MethodInfo of a
method
            // for an assembly loaded into the TestAssemblyLoadContext in a
static variable.
            entryPoint = new TestInfo(a.EntryPoint);
            testEntryPoint = a.EntryPoint;

            var oResult = a.EntryPoint?.Invoke(null, args);
            alc.Unload();
            return (oResult is int result) ? result : -1;
        }

        static void Main(string[] args)
        {
            WeakReference testAlcWeakRef;
            // Issue preventing unloading #2 - we keep MethodInfo of a
method for an assembly loaded into the TestAssemblyLoadContext in a local
variable
            MethodInfo? testEntryPoint;
            int result = ExecuteAndUnload(@"absolute/path/to/test.dll", out
testAlcWeakRef, out testEntryPoint);

            for (int i = 0; testAlcWeakRef.IsAlive && (i < 10); i++)
            {
                GC.Collect();
                GC.WaitForPendingFinalizers();
            }

            System.Diagnostics.Debugger.Break();
        }
    }
}
```

```
        Console.WriteLine($"Test completed, result={result}, entryPoint:  
{testEntryPoint} unload success: {!testAlcWeakRef.IsAlive}");  
    }  
}  
}
```

Program loaded into the TestAssemblyLoadContext

The following code represents the *test.dll* passed to the `ExecuteAndUnload` method in the main testing program.

C#

```
using System;  
using System.Runtime.InteropServices;  
using System.Threading;  
  
namespace test  
{  
    class Test  
    {  
    }  
  
    class Program  
    {  
        public static void ThreadProc()  
        {  
            // Issue preventing unloading #4 - a thread running method  
            // inside of the TestAssemblyLoadContext at the unload time  
            Thread.Sleep(Timeout.Infinite);  
        }  
  
        static GCHandle handle;  
        static int Main(string[] args)  
        {  
            // Issue preventing unloading #3 - normal GC handle  
            handle = GCHandle.Alloc(new Test());  
            Thread t = new Thread(new ThreadStart(ThreadProc));  
            t.IsBackground = true;  
            t.Start();  
            Console.WriteLine($"Hello from the test: args[0] = {args[0]}");  
  
            return 1;  
        }  
    }  
}
```

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

Reference assemblies

Article • 09/15/2021

Reference assemblies are a special type of assembly that contain only the minimum amount of metadata required to represent the library's public API surface. They include declarations for all members that are significant when referencing an assembly in build tools, but exclude all member implementations and declarations of private members that have no observable impact on their API contract. In contrast, regular assemblies are called *implementation assemblies*.

Reference assemblies can't be loaded for execution, but they can be passed as compiler input in the same way as implementation assemblies. Reference assemblies are usually distributed with the Software Development Kit (SDK) of a particular platform or library.

Using a reference assembly enables developers to build programs that target a specific library version without having the full implementation assembly for that version.

Suppose, you have only the latest version of some library on your machine, but you want to build a program that targets an earlier version of that library. If you compile directly against the implementation assembly, you might inadvertently use API members that aren't available in the earlier version. You'll only find this mistake when testing the program on the target machine. If you compile against the reference assembly for the earlier version, you'll immediately get a compile-time error.

A reference assembly can also represent a contract, that is, a set of APIs that don't correspond to the concrete implementation assembly. Such reference assemblies, called the *contract assembly*, can be used to target multiple platforms that support the same set of APIs. For example, .NET Standard provides the contract assembly, *netstandard.dll*, that represents the set of common APIs shared between different .NET platforms. The implementations of these APIs are contained in different assemblies on different platforms, such as *mscorlib.dll* on .NET Framework or *System.Private.CoreLib.dll* on .NET Core. A library that targets .NET Standard can run on all platforms that support .NET Standard.

Using reference assemblies

To use certain APIs from your project, you must add references to their assemblies. You can add references to either implementation assemblies or to reference assemblies. It's recommended you use reference assemblies whenever they're available. Doing so ensures that you're using only the supported API members in the target version, meant

to be used by API designers. Using the reference assembly ensures you're not taking a dependency on implementation details.

Reference assemblies for the .NET Framework libraries are distributed with targeting packs. You can obtain them by downloading a standalone installer or by selecting a component in Visual Studio installer. For more information, see [Install the .NET Framework for developers](#). For .NET Core and .NET Standard, reference assemblies are automatically downloaded as necessary (via NuGet) and referenced. For .NET Core 3.0 and higher, the reference assemblies for the core framework are in the [Microsoft.NETCore.App.Ref](#) package (the [Microsoft.NETCore.App](#) package is used instead for versions before 3.0).

When you add references to .NET Framework assemblies in Visual Studio using the [Add reference](#) dialog, you select an assembly from the list, and Visual Studio automatically finds reference assemblies that correspond to the target framework version selected in your project. The same applies to adding references directly into MSBuild project using the [Reference](#) project item: you only need to specify the assembly name, not the full file path. When you add references to these assemblies in the command line by using the `-reference` compiler option ([in C#](#) and in [Visual Basic](#)) or by using the [Compilation.AddReferences](#) method in the Roslyn API, you must manually specify reference assembly files for the correct target platform version. .NET Framework reference assembly files are located in the `%ProgramFiles(x86)%\Reference Assemblies\Microsoft\Framework\.NETFramework` directory. For .NET Core, you can force publish operation to copy reference assemblies for your target platform into the `publish/refs` subdirectory of your output directory by setting the `PreserveCompilationContext` project property to `true`. Then you can pass these reference assembly files to the compiler. Using `DependencyContext` from [Microsoft.Extensions.DependencyModel](#) package can help locate their paths.

Because they contain no implementation, reference assemblies can't be loaded for execution. Trying to do so results in a [System.BadImageFormatException](#). If you want to examine the contents of a reference assembly, you can load it into the reflection-only context in .NET Framework (using the [Assembly.ReflectionOnlyLoad](#) method) or into the [MetadataLoadContext](#) in .NET Core.

Generating reference assemblies

Generating reference assemblies for your libraries can be useful when your library consumers need to build their programs against many different versions of the library. Distributing implementation assemblies for all these versions might be impractical

because of their large size. Reference assemblies are smaller in size, and distributing them as a part of your library's SDK reduces download size and saves disk space.

IDEs and build tools also can take advantage of reference assemblies to reduce build times in case of large solutions consisting of multiple class libraries. Usually, in incremental build scenarios a project is rebuilt when any of its input files are changed, including the assemblies it depends on. The implementation assembly changes whenever the programmer changes the implementation of any member. The reference assembly only changes when its public API is affected. So, using the reference assembly as an input file instead of the implementation assembly allows skipping the build of the dependent project in some cases.

You can generate reference assemblies:

- In an MSBuild project, by using the [ProduceReferenceAssembly](#) project property.
- When compiling program from command line, by specifying `-refonly` ([C# / Visual Basic](#)) or `-refout` ([C# / Visual Basic](#)) compiler options.
- When using the Roslyn API, by setting `EmitOptions.EmitMetadataOnly` to `true` and `EmitOptions.IncludePrivateMembers` to `false` in an object passed to the `Compilation.Emit` method.

If you want to distribute reference assemblies with NuGet packages, you must include them in the `ref\` subdirectory under the package directory instead of in the `lib\` subdirectory used for implementation assemblies.

Reference assemblies structure

Reference assemblies are an expansion of the related concept, *metadata-only assemblies*. Metadata-only assemblies have their method bodies replaced with a single `throw null` body, but include all members except anonymous types. The reason for using `throw null` bodies (as opposed to no bodies) is so that `PEVerify` can run and pass (thus validating the completeness of the metadata).

Reference assemblies further remove metadata (private members) from metadata-only assemblies:

- A reference assembly only has references for what it needs in the API surface. The real assembly may have additional references related to specific implementations. For instance, the reference assembly for `class C { private void M() { dynamic d = 1; ... } }` doesn't reference any types required for `dynamic`.
- Private function-members (methods, properties, and events) are removed in cases where their removal doesn't observably impact compilation. If there are no

`InternalsVisibleTo` attributes, internal function members are also removed.

The metadata in reference assemblies continues to keep the following information:

- All types, including private and nested types.
- All attributes, even internal ones.
- All virtual methods.
- Explicit interface implementations.
- Explicitly implemented properties and events, because their accessors are virtual.
- All fields of structures.

Reference assemblies include an assembly-level `ReferenceAssembly` attribute. This attribute may be specified in source; then the compiler won't need to synthesize it. Because of this attribute, runtimes will refuse to load reference assemblies for execution (but they can be loaded in reflection-only mode).

Exact reference assembly structure details depend on the compiler version. Newer versions may choose to exclude more metadata if it's determined as not affecting the public API surface.

Note

Information in this section is applicable only to reference assemblies generated by Roslyn compilers starting from C# version 7.1 or Visual Basic version 15.3. The structure of reference assemblies for .NET Framework and .NET Core libraries can differ in some details, because they use their own mechanism of generating reference assemblies. For example, they might have totally empty method bodies instead of the `throw null` body. But the general principle still applies: they don't have usable method implementations and contain metadata only for members that have an observable impact from a public API perspective.

See also

- [Assemblies in .NET](#)
- [Framework targeting overview](#)
- [How to: Add or remove references by using the Reference Manager](#)



Collaborate with us on
GitHub

.NET

.NET feedback

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

Resolve assembly loads

Article • 09/15/2021

.NET provides the [AppDomain.AssemblyResolve](#) event for applications that require greater control over assembly loading. By handling this event, your application can load an assembly into the load context from outside the normal probing paths, select which of several assembly versions to load, emit a dynamic assembly and return it, and so on. This topic provides guidance for handling the [AssemblyResolve](#) event.

ⓘ Note

For resolving assembly loads in the reflection-only context, use the [AppDomain.ReflectionOnlyAssemblyResolve](#) event instead.

How the AssemblyResolve event works

When you register a handler for the [AssemblyResolve](#) event, the handler is invoked whenever the runtime fails to bind to an assembly by name. For example, calling the following methods from user code can cause the [AssemblyResolve](#) event to be raised:

- An [AppDomain.Load](#) method overload or [Assembly.Load](#) method overload whose first argument is a string that represents the display name of the assembly to load (that is, the string returned by the [Assembly.FullName](#) property).
- An [AppDomain.Load](#) method overload or [Assembly.Load](#) method overload whose first argument is an [AssemblyName](#) object that identifies the assembly to load.
- An [Assembly.LoadWithPartialName](#) method overload.
- An [AppDomain.CreateInstance](#) or [AppDomain.CreateInstanceAndUnwrap](#) method overload that instantiates an object in another application domain.

What the event handler does

The handler for the [AssemblyResolve](#) event receives the display name of the assembly to be loaded, in the [ResolveEventArgs.Name](#) property. If the handler does not recognize the assembly name, it returns `null` (C#), `Nothing` (Visual Basic), or `nullptr` (Visual C++).

If the handler recognizes the assembly name, it can load and return an assembly that satisfies the request. The following list describes some sample scenarios.

- If the handler knows the location of a version of the assembly, it can load the assembly by using the [Assembly.LoadFrom](#) or [Assembly.LoadFile](#) method, and can return the loaded assembly if successful.
- If the handler has access to a database of assemblies stored as byte arrays, it can load a byte array by using one of the [Assembly.Load](#) method overloads that take a byte array.
- The handler can generate a dynamic assembly and return it.

 **Note**

The handler must load the assembly into the load-from context, into the load context, or without context. If the handler loads the assembly into the reflection-only context by using the [Assembly.ReflectionOnlyLoad](#) or the [Assembly.ReflectionOnlyLoadFrom](#) method, the load attempt that raised the [AssemblyResolve](#) event fails.

It is the responsibility of the event handler to return a suitable assembly. The handler can parse the display name of the requested assembly by passing the [ResolveEventArgs.Name](#) property value to the [AssemblyName\(String\)](#) constructor. Beginning with the .NET Framework 4, the handler can use the [ResolveEventArgs.RequestingAssembly](#) property to determine whether the current request is a dependency of another assembly. This information can help identify an assembly that will satisfy the dependency.

The event handler can return a different version of the assembly than the version that was requested.

In most cases, the assembly that is returned by the handler appears in the load context, regardless of the context the handler loads it into. For example, if the handler uses the [Assembly.LoadFrom](#) method to load an assembly into the load-from context, the assembly appears in the load context when the handler returns it. However, in the following case the assembly appears without context when the handler returns it:

- The handler loads an assembly without context.
- The [ResolveEventArgs.RequestingAssembly](#) property is not null.
- The requesting assembly (that is, the assembly that is returned by the [ResolveEventArgs.RequestingAssembly](#) property) was loaded without context.

For information about contexts, see the [Assembly.LoadFrom\(String\)](#) method overload.

Multiple versions of the same assembly can be loaded into the same application domain. This practice is not recommended, because it can lead to type assignment problems. See [Best practices for assembly loading](#).

What the event handler should not do

The primary rule for handling the [AssemblyResolve](#) event is that you should not try to return an assembly you do not recognize. When you write the handler, you should know which assemblies might cause the event to be raised. Your handler should return null for other assemblies.

Important

Beginning with the .NET Framework 4, the [AssemblyResolve](#) event is raised for satellite assemblies. This change affects an event handler that was written for an earlier version of the .NET Framework, if the handler tries to resolve all assembly load requests. Event handlers that ignore assemblies they do not recognize are not affected by this change: They return `null`, and normal fallback mechanisms are followed.

When loading an assembly, the event handler must not use any of the [AppDomain.Load](#) or [Assembly.Load](#) method overloads that can cause the [AssemblyResolve](#) event to be raised recursively, because this can lead to a stack overflow. (See the list provided earlier in this topic.) This happens even if you provide exception handling for the load request, because no exception is thrown until all event handlers have returned. Thus, the following code results in a stack overflow if `MyAssembly` is not found:

C#

```
using System;
using System.Reflection;

class BadExample
{
    static void Main()
    {
        AppDomain ad = AppDomain.CreateDomain("Test");
        ad.AssemblyResolve += MyHandler;

        try
        {
            object obj = ad.CreateInstanceAndUnwrap(
                "MyAssembly, version=1.2.3.4, culture=neutral,
                publicKeyToken=null",
```

```

        "MyType");
    }
    catch (Exception ex)
    {
        Console.WriteLine(ex.Message);
    }
}

static Assembly MyHandler(object source, ResolveEventArgs e)
{
    Console.WriteLine("Resolving {0}", e.Name);
    // DO NOT DO THIS: This causes a StackOverflowException
    return Assembly.Load(e.Name);
}
}

/* This example produces output similar to the following:

Resolving MyAssembly, Version=1.2.3.4, Culture=neutral, PublicKeyToken=null
Resolving MyAssembly, Version=1.2.3.4, Culture=neutral, PublicKeyToken=null
...
Resolving MyAssembly, Version=1.2.3.4, Culture=neutral, PublicKeyToken=null
Resolving MyAssembly, Version=1.2.3.4, Culture=neutral, PublicKeyToken=null

Process is terminated due to StackOverflowException.
*/

```

The correct way to handle AssemblyResolve

When resolving assemblies from the `AssemblyResolve` event handler, a `StackOverflowException` will eventually be thrown if the handler uses the `Assembly.Load` or `AppDomain.Load` method calls. Instead, use `LoadFile` or `LoadFrom` methods, as they do not raise the `AssemblyResolve` event.

Imagine that `MyAssembly.dll` is located near the executing assembly, in a known location, it can be resolved using `Assembly.LoadFile` given the path to the assembly.

C#

```

using System;
using System.IO;
using System.Reflection;

class CorrectExample
{
    static void Main()
    {
        AppDomain ad = AppDomain.CreateDomain("Test");
        ad.AssemblyResolve += MyHandler;
    }
}

static Assembly MyHandler(object source, ResolveEventArgs e)
{
    Console.WriteLine("Resolving {0}", e.Name);
    // DO NOT DO THIS: This causes a StackOverflowException
    return Assembly.Load(e.Name);
}

```

```
try
{
    object obj = ad.CreateInstanceAndUnwrap(
        "MyAssembly", version=1.2.3.4, culture=neutral,
publicKeyToken=null",
        "MyType");
}
catch (Exception ex)
{
    Console.WriteLine(ex.Message);
}
}

static Assembly MyHandler(object source, ResolveEventArgs e)
{
    Console.WriteLine("Resolving {0}", e.Name);

    var path = Path.GetFullPath("../..\\MyAssembly.dll");
    return Assembly.LoadFile(path);
}
}
```

See also

- [Best practices for assembly loading](#)
- [Use application domains](#)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



.NET feedback

The .NET documentation is open source. Provide feedback [here](#).

 [Open a documentation issue](#)

 [Provide product feedback](#)

Create assemblies

Article • 09/15/2021

You can create single-file or multifile assemblies using an IDE, such as Visual Studio, or the compilers and tools provided by the Windows SDK. The simplest assembly is a single file that has a simple name and is loaded into a single application domain. This assembly cannot be referenced by other assemblies outside the application directory and does not undergo version checking. To uninstall the application made up of the assembly, you simply delete the directory where it resides. For many developers, an assembly with these features is all that is needed to deploy an application.

You can create a multifile assembly from several code modules and resource files. You can also create an assembly that can be shared by multiple applications. A shared assembly must have a strong name and can be deployed in the global assembly cache.

You have several options when grouping code modules and resources into assemblies, depending on the following factors:

- **Versioning**

Group modules that should have the same version information.

- **Deployment**

Group code modules and resources that support your model of deployment.

- **Reuse**

Group modules if they can be logically used together for some purpose. For example, an assembly consisting of types and classes used infrequently for program maintenance can be put in the same assembly. In addition, types that you intend to share with multiple applications should be grouped into an assembly and the assembly should be signed with a strong name.

- **Security**

Group modules containing types that require the same security permissions.

- **Scoping**

Group modules containing types whose visibility should be restricted to the same assembly.

There are special considerations when making common language runtime assemblies available to unmanaged COM applications. For more information about working with unmanaged code, see [Expose .NET Framework components to COM](#).

See also

- [Assembly versioning](#)
- [How to: Build a single-file assembly](#)
- [How to: Build a multifile assembly](#)
- [How the runtime locates assemblies](#)
- [Multifile assemblies](#)

 [Collaborate with us on GitHub](#)

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

 .NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

Assembly names

Article • 09/15/2021

An assembly's name is stored in metadata and has a significant impact on the assembly's scope and use by an application. A strong-named assembly has a fully qualified name that includes the assembly's name, culture, public key, version number, and, optionally, processor architecture. Use the [FullName](#) property to obtain the fully qualified name, frequently referred to as the display name, for loaded assemblies.

The runtime uses the name information to locate the assembly and differentiate it from other assemblies with the same name. For example, a strong-named assembly called `myTypes` could have the following fully qualified name:

```
myTypes, Version=1.0.1234.0, Culture=en-US,  
PublicKeyToken=b77a5c561934e089c, ProcessorArchitecture=msil
```

In this example, the fully qualified name indicates that the `myTypes` assembly has a strong name with a public key token, has the culture value for United States English, and has a version number of 1.0.1234.0. Its processor architecture is `msil`, which means that it will be just-in-time (JIT)-compiled to 32-bit code or 64-bit code depending on the operating system and processor.

Tip

The `ProcessorArchitecture` information allows processor-specific versions of assemblies. You can create versions of an assembly whose identity differs only by processor architecture, for example 32-bit and 64-bit processor-specific versions. Processor architecture is not required for strong names. For more information, see [AssemblyName.ProcessorArchitecture](#).

Code that requests types in an assembly must use a fully qualified assembly name. This is called fully qualified binding. Partial binding, which specifies only an assembly name, is not permitted when referencing assemblies in .NET Framework.

All assembly references to assemblies that make up .NET Framework must also contain the fully qualified name of the assembly. For example, a reference to the `System.Data` .NET Framework assembly for version 1.0 would include:

```
System.data, version=1.0.3300.0, Culture=neutral,  
PublicKeyToken=b77a5c561934e089
```

The version corresponds to the version number of all .NET Framework assemblies that shipped with .NET Framework version 1.0. For .NET Framework assemblies, the culture value is always neutral, and the public key is the same as shown in the above example.

For example, to add an assembly reference in a configuration file to set up a trace listener, you would include the fully qualified name of the system .NET Framework assembly:

XML

```
<add name="myListener" type="System.Diagnostics.TextWriterTraceListener,  
System, Version=1.0.3300.0, Culture=neutral,  
PublicKeyToken=b77a5c561934e089" initializeData="c:\myListener.log" />
```

ⓘ Note

The runtime treats assembly names as case-insensitive when binding to an assembly, but preserves whatever case is used in an assembly name. Several tools in the Windows SDK handle assembly names as case-sensitive. For best results, manage assembly names as though they were case-sensitive.

Name application components

The runtime does not consider the file name when determining an assembly's identity. The assembly identity, which consists of the assembly name, version, culture, and strong name, must be clear to the runtime.

For example, if you have an assembly called *myAssembly.exe* that references an assembly called *myAssembly.dll*, binding occurs correctly if you execute *myAssembly.exe*. However, if another application executes *myAssembly.exe* using the method [AppDomain.ExecuteAssembly](#), the runtime determines that `myAssembly` is already loaded when *myAssembly.exe* requests binding to `myAssembly`. In this case, *myAssembly.dll* is never loaded. Because *myAssembly.exe* does not contain the requested type, a [TypeLoadException](#) occurs.

To avoid this problem, make sure the assemblies that make up your application do not have the same assembly name or place assemblies with the same name in different directories.

Note

In .NET Framework, if you put a strong-named assembly in the global assembly cache, the assembly's file name must match the assembly name, not including the file name extension, such as `.exe` or `.dll`. For example, if the file name of an assembly is `myAssembly.dll`, the assembly name must be `myAssembly`. Private assemblies deployed only in the root application directory can have an assembly name that is different from the file name.

See also

- [How to: Determine an assembly's fully qualified name](#)
- [Create assemblies](#)
- [Strong-named assemblies](#)
- [Global assembly cache](#)
- [How the runtime locates assemblies](#)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

How to: Find an assembly's fully qualified name

Article • 09/15/2021

To discover the fully qualified name of a .NET Framework assembly in the global assembly cache, use the Global Assembly Cache tool ([Gacutil.exe](#)). See [How to: View the contents of the global assembly cache](#).

For .NET Core assemblies, and for .NET Framework assemblies that aren't in the global assembly cache, you can get the fully qualified assembly name in a number of ways:

- You can use code to output the information to the console or to a variable, or you can use the [Ildasm.exe \(IL Disassembler\)](#) to examine the assembly's metadata, which contains the fully qualified name.
- If the assembly is already loaded by the application, you can retrieve the value of the [Assembly.FullName](#) property to get the fully qualified name. You can use the [Assembly](#) property of a [Type](#) defined in that assembly to retrieve a reference to the [Assembly](#) object. The example provides an illustration.
- If you know the assembly's file system path, you can call the `static` (C#) or `shared` (Visual Basic) [AssemblyName.GetAssemblyName](#) method to get the fully qualified assembly name. The following is a simple example.

C#

```
using System;
using System.Reflection;

public class Example
{
    public static void Main()
    {

        Console.WriteLine(AssemblyName.GetAssemblyName(@".\UtilityLibrary.dll"))
    }
}

// The example displays output like the following:
// UtilityLibrary, Version=1.1.0.0, Culture=neutral,
// PublicKeyToken=null
```

- You can use the [Ildasm.exe \(IL Disassembler\)](#) to examine the assembly's metadata, which contains the fully qualified name.

For more information about setting assembly attributes such as version, culture, and assembly name, see [Set assembly attributes](#). For more information about giving an assembly a strong name, see [Create and use strong-named assemblies](#).

Example

The following example shows how to display the fully qualified name of an assembly containing a specified class to the console. It uses the [Type.Assembly](#) property to retrieve a reference to an assembly from a type that's defined in that assembly.

C#

```
using System;
using System.Reflection;

class asmname
{
    public static void Main()
    {
        Type t = typeof(System.Data.DataSet);
        string s = t.Assembly.FullName.ToString();
        Console.WriteLine("The fully qualified assembly name " +
            "containing the specified class is {0}.", s);
    }
}
```

See also

- [Assembly names](#)
- [Create assemblies](#)
- [Create and use strong-named assemblies](#)
- [Global assembly cache](#)
- [How the runtime locates assemblies](#)

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

more information, see [our contributor guide](#).

Assembly location

Article • 09/15/2021

An assembly's location determines whether the common language runtime can locate it when referenced, and can also determine whether the assembly can be shared with other assemblies. You can deploy an assembly in the following locations:

- The application's directory or subdirectories.

This is the most common location for deploying an assembly. The subdirectories of an application's root directory can be based on language or culture. If an assembly has information in the culture attribute, it must be in a subdirectory under the application directory with that culture's name.

- The global assembly cache.

This is a machine-wide code cache that is installed wherever the common language runtime is installed. In most cases, if you intend to share an assembly with multiple applications, you should deploy it into the global assembly cache.

- On an HTTP server.

An assembly deployed on an HTTP server must have a strong name; you point to the assembly in the codebase section of the application's configuration file.

See also

- [Create assemblies](#)
- [Global assembly cache](#)
- [How the runtime locates assemblies](#)

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

Set assembly attributes

Article • 01/18/2023

Assembly attributes are values that provide information about an assembly. They're usually set in an `AssemblyInfo.cs` file. The attributes are divided into the following sets of information:

- Assembly identity attributes
- Informational attributes
- Assembly manifest attributes
- Strong name attributes

Tip

This article is scoped to adding assembly attributes from code. For information on adding assembly attributes to projects (not in code), see [MSBuild reference for .NET SDK projects: Assembly attribute properties](#).

Assembly identity attributes

Three attributes, together with a strong name (if applicable), determine the identity of an assembly: name, version, and culture. These attributes form the full name of the assembly and are required when referencing the assembly in code. You can use attributes to set an assembly's version and culture. The compiler or the [Assembly Linker \(Al.exe\)](#) sets the name value when the assembly is created, based on the file containing the assembly manifest.

The following table describes the version and culture attributes.

| Assembly identity attribute | Description |
|--|--|
| AssemblyCultureAttribute | Enumerated field indicating the culture that the assembly supports. An assembly can also specify culture independence, indicating that it contains the resources for the default culture. Note: The runtime treats any assembly that does not have the culture attribute set to null as a satellite assembly. Such assemblies are subject to satellite assembly binding rules. For more information, see How the runtime locates assemblies . |
| AssemblyFlagsAttribute | Value that sets assembly attributes, such as whether the assembly can be run side by side. |

| Assembly identity attribute | Description |
|--|---|
| AssemblyVersionAttribute | Numeric value in the format <i>major.minor.build.revision</i> (for example, 2.4.0.0). The common language runtime uses this value to perform binding operations in strong-named assemblies. Note: If the AssemblyInformationalVersionAttribute attribute is not applied to an assembly, the version number specified by the AssemblyVersionAttribute attribute is used by the Application.ProductVersion , Application.UserAppDataPath , and Application.UserAppDataRegistry properties. |

The following code example shows how to apply the version and culture attributes to an assembly.

C#

```
// Set version number for the assembly.
[assembly:AssemblyVersionAttribute("4.3.2.1")]
// Set culture as German.
[assembly:AssemblyCultureAttribute("de")]
```

Informational attributes

You can use informational attributes to provide additional company or product information for an assembly. The following table describes the informational attributes you can apply to an assembly.

| Informational attribute | Description |
|---|--|
| AssemblyCompanyAttribute | String value specifying a company name. |
| AssemblyCopyrightAttribute | String value specifying copyright information. |
| AssemblyFileVersionAttribute | String value specifying the Win32 file version number. This normally defaults to the assembly version. |
| AssemblyInformationalVersionAttribute | String value specifying version information that is not used by the common language runtime, such as a full product version number. Note: If this attribute is applied to an assembly, the string it specifies can be obtained at run time by using the Application.ProductVersion property. The string is also used in the path and registry key provided by the Application.UserAppDataPath and Application.UserAppDataRegistry properties. |

| Informational attribute | Description |
|--|--|
| AssemblyProductAttribute | String value specifying product information. |
| AssemblyTrademarkAttribute | String value specifying trademark information. |

These attributes can appear on the Windows Properties page of the assembly, or they can be overridden using the `/win32res` compiler option to specify your Win32 resource file.

Assembly manifest attributes

You can use assembly manifest attributes to provide information in the assembly manifest, including title, description, the default alias, and configuration. The following table describes the assembly manifest attributes.

| Assembly manifest attribute | Description |
|--|---|
| AssemblyConfigurationAttribute | String value indicating the configuration of the assembly, such as Retail or Debug. The runtime does not use this value. |
| AssemblyDefaultAliasAttribute | String value specifying a default alias to be used by referencing assemblies. This value provides a friendly name when the name of the assembly itself is not friendly (such as a GUID value). This value can also be used as a short form of the full assembly name. |
| AssemblyDescriptionAttribute | String value specifying a short description that summarizes the nature and purpose of the assembly. |
| AssemblyTitleAttribute | String value specifying a friendly name for the assembly. For example, an assembly named <code>comdlg</code> might have the title Microsoft Common Dialog Control. |

Strong name attributes

You can use strong name attributes to set a strong name for an assembly. The following table describes the strong name attributes.

| Strong name attribute | Description |
|--|---|
| AssemblyDelaySignAttribute | Boolean value indicating that delay signing is being used. |
| AssemblyKeyFileAttribute | String value indicating the name of the file that contains either the public key (if using delay signing) or both the public and private keys passed as a parameter to the constructor of this attribute. |

| Strong name attribute | Description |
|--|--|
| | Note that the file name is relative to the output file path (the <code>.exe</code> or <code>.dll</code>), not the source file path. |
| AssemblyKeyNameAttribute | Indicates the key container that contains the key pair passed as a parameter to the constructor of this attribute. |

The following code example shows the attributes to apply when using delay signing to create a strong-named assembly with a public key file called `myKey.snk`.

C#

```
[assembly:AssemblyKeyFileAttribute("myKey.snk")]
[assembly:AssemblyDelaySignAttribute(true)]
```

See also

- [Create assemblies](#)
- [MSBuild reference for .NET SDK projects](#)

 [Collaborate with us on GitHub](#)

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

 .NET

[.NET feedback](#)

The .NET documentation is open source. Provide feedback [here](#).

 [Open a documentation issue](#)

 [Provide product feedback](#)

Strong-named assemblies

Article • 09/15/2021

Strong-naming an assembly creates a unique identity for the assembly, and can prevent assembly conflicts.

What makes a strong-named assembly?

A strong named assembly is generated by using the private key that corresponds to the public key distributed with the assembly, and the assembly itself. The assembly includes the assembly manifest, which contains the names and hashes of all the files that make up the assembly. Assemblies that have the same strong name should be identical.

You can strong-name assemblies by using Visual Studio or a command-line tool. For more information, see [How to: Sign an assembly with a strong name](#) or [Sn.exe \(Strong Name tool\)](#).

When a strong-named assembly is created, it contains the simple text name of the assembly, the version number, optional culture information, a digital signature, and the public key that corresponds to the private key used for signing.

⚠ Warning

Do not rely on strong names for security. They provide a unique identity only.

Why strong-name your assemblies?

For .NET Framework, strong-named assemblies are useful in the following scenarios:

- You want to enable your assemblies to be referenced by strong-named assemblies, or you want to give `friend` access to your assemblies from other strong-named assemblies.
- An app needs access to different versions of the same assembly. This means you need different versions of an assembly to load side by side in the same app domain without conflict. For example, if different extensions of an API exist in assemblies that have the same simple name, strong-naming provides a unique identity for each version of the assembly.

- You do not want to negatively affect performance of apps using your assembly, so you want the assembly to be domain neutral. This requires strong-naming because a domain-neutral assembly must be installed in the global assembly cache.
- You want to centralize servicing for your app by applying publisher policy, which means the assembly must be installed in the global assembly cache.

For .NET Core and .NET 5+, strong-named assemblies do not provide material benefits. The runtime never validates the strong-name signature, nor does it use the strong-name for assembly binding.

If you are an open-source developer and you want the identity benefits of a strong-named assembly for better compatibility with .NET Framework, consider checking in the private key associated with an assembly to your source control system.

See also

- [Global assembly cache](#)
- [How to: Sign an assembly with a strong name](#)
- [Sn.exe \(Strong Name tool\)](#)
- [Create and use strong-named assemblies](#)

 [Collaborate with us on GitHub](#)

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

Create and use strong-named assemblies

Article • 09/15/2021

A strong name consists of the assembly's identity—its simple text name, version number, and culture information (if provided)—plus a public key and a digital signature. It is generated from an assembly file using the corresponding private key. (The assembly file contains the assembly manifest, which contains the names and hashes of all the files that make up the assembly.)

⚠ Warning

Do not rely on strong names for security. They provide a unique identity only.

A strong-named assembly can only use types from other strong-named assemblies. Otherwise, the integrity of the strong-named assembly would be compromised.

ⓘ Note

Although .NET Core supports strong-named assemblies, and all assemblies in the .NET Core library are signed, the majority of third-party assemblies do not need strong names. For more information, see [Strong Name Signing](#) on GitHub.

Strong name scenario

The following scenario outlines the process of signing an assembly with a strong name and later referencing it by that name.

1. Assembly A is created with a strong name using one of the following methods:

- Using a development environment that supports creating strong names, such as Visual Studio.
- Creating a cryptographic key pair using the [Strong Name tool \(Sn.exe\)](#) and assigning that key pair to the assembly using either a command-line compiler or the [Assembly Linker \(Al.exe\)](#). The Windows SDK provides both Sn.exe and Al.exe.

2. The development environment or tool signs the hash of the file containing the assembly's manifest with the developer's private key. This digital signature is stored in the portable executable (PE) file that contains Assembly A's manifest.
3. Assembly B is a consumer of Assembly A. The reference section of Assembly B's manifest includes a token that represents Assembly A's public key. A token is a portion of the full public key and is used rather than the key itself to save space.
4. The common language runtime verifies the strong name signature when the assembly is placed in the global assembly cache. When binding by strong name at run time, the common language runtime compares the key stored in Assembly B's manifest with the key used to generate the strong name for Assembly A. If the .NET security checks pass and the bind succeeds, Assembly B has a guarantee that Assembly A's bits have not been tampered with and that these bits actually come from the developers of Assembly A.

 **Note**

This scenario doesn't address trust issues. Assemblies can carry full Microsoft Authenticode signatures in addition to a strong name. Authenticode signatures include a certificate that establishes trust. It's important to note that strong names don't require code to be signed in this way. Strong names only provide a unique identity.

Bypass signature verification of trusted assemblies

Starting with the .NET Framework 3.5 Service Pack 1, strong-name signatures are not validated when an assembly is loaded into a full-trust application domain, such as the default application domain for the `MyComputer` zone. This is referred to as the strong-name bypass feature. In a full-trust environment, demands for [StrongNameIdentityPermission](#) always succeed for signed, full-trust assemblies, regardless of their signature. The strong-name bypass feature avoids the unnecessary overhead of strong-name signature verification of full-trust assemblies in this situation, allowing the assemblies to load faster.

The bypass feature applies to any assembly that is signed with a strong name and that has the following characteristics:

- Fully trusted without [StrongName](#) evidence (for example, has `MyComputer` zone evidence).
- Loaded into a fully trusted [AppDomain](#).
- Loaded from a location under the [ApplicationBase](#) property of that [AppDomain](#).
- Not delay-signed.

This feature can be disabled for individual applications or for a computer. See [How to: Disable the strong-name bypass feature](#).

Related topics

| Title | Description |
|--|--|
| How to: Create a public-private key pair | Describes how to create a cryptographic key pair for signing an assembly. |
| How to: Sign an assembly with a strong name | Describes how to create a strong-named assembly. |
| Enhanced strong naming | Describes enhancements to strong-names in the .NET Framework 4.5. |
| How to: Reference a strong-named assembly | Describes how to reference types or resources in a strong-named assembly at compile time or run time. |
| How to: Disable the strong-name bypass feature | Describes how to disable the feature that bypasses the validation of strong-name signatures. This feature can be disabled for all or for specific applications. |
| Create assemblies | Provides an overview of single-file and multifile assemblies. |
| How to delay sign an assembly in Visual Studio | Explains how to sign an assembly with a strong name after the assembly has been created. |
| Sn.exe (Strong Name tool) | Describes the tool included in the .NET Framework that helps create assemblies with strong names. This tool provides options for key management, signature generation, and signature verification. |
| Al.exe (Assembly linker) | Describes the tool included in the .NET Framework that generates a file that has an assembly manifest from modules or resource files. |

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

How to: Create a public-private key pair

Article • 09/15/2021

To sign an assembly with a strong name, you must have a public/private key pair. This public and private cryptographic key pair is used during compilation to create a strong-named assembly. You can create a key pair using the [Strong Name tool \(Sn.exe\)](#). Key pair files usually have an *.snk* extension.

ⓘ Note

In Visual Studio, the C# and Visual Basic project property pages include a **Signing** tab that enables you to select existing key files or to generate new key files without using *Sn.exe*. In Visual C++, you can specify the location of an existing key file in the **Advanced** property page in the **Linker** section of the **Configuration Properties** section of the **Property Pages** window. The use of the **AssemblyKeyFileAttribute** attribute to identify key file pairs was made obsolete beginning with Visual Studio 2005.

Create a key pair

To create a key pair, at a command prompt, type the following command:

```
sn -k <file name>
```

In this command, *file name* is the name of the output file containing the key pair.

The following example creates a key pair called *sgKey.snk*.

```
Windows Command Prompt
```

```
sn -k sgKey.snk
```

If you intend to delay sign an assembly and you control the whole key pair (which is unlikely outside test scenarios), you can use the following commands to generate a key pair and then extract the public key from it into a separate file. First, create the key pair:

```
Windows Command Prompt
```

```
sn -k keypair.snk
```

Next, extract the public key from the key pair and copy it to a separate file:

```
Windows Command Prompt
```

```
sn -p keypair.snk public.snk
```

Once you create the key pair, you must put the file where the strong name signing tools can find it.

When signing an assembly with a strong name, the [Assembly Linker \(Al.exe\)](#) looks for the key file relative to the current directory and to the output directory. When using command-line compilers, you can simply copy the key to the current directory containing your code modules.

If you are using an earlier version of Visual Studio that does not have a **Signing** tab in the project properties, the recommended key file location is the project directory with the file attribute specified as follows:

```
C#
```

```
[assembly:AssemblyKeyFileAttribute("keyfile.snk")]
```

See also

- [Create and use strong-named assemblies](#)

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

How to: Sign an assembly with a strong name

Article • 08/31/2022

ⓘ Note

Although .NET Core supports strong-named assemblies, and all assemblies in the .NET Core library are signed, the majority of third-party assemblies do not need strong names. For more information, see [Strong Name Signing](#) on GitHub.

There are a number of ways to sign an assembly with a strong name:

- By using the [Build > Strong naming](#) page in the [project designer](#) for a project in Visual Studio. This is the easiest and most convenient way to sign an assembly with a strong name.
- By using the [Assembly Linker \(Al.exe\)](#) to link a .NET Framework code module (a *.netmodule* file) with a key file.
- By using assembly attributes to insert the strong name information into your code. You can use either the [AssemblyKeyFileAttribute](#) or the [AssemblyKeyNameAttribute](#) attribute, depending on where the key file to be used is located.
- By using compiler options.

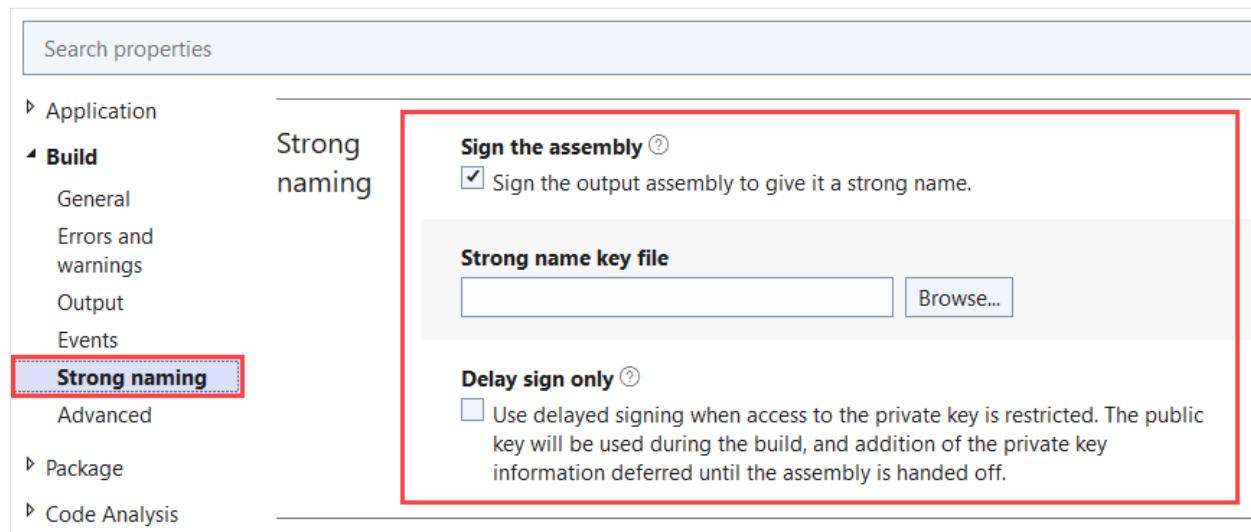
You must have a cryptographic key pair to sign an assembly with a strong name. For more information about creating a key pair, see [How to: Create a public-private key pair](#).

Create and sign an assembly with a strong name by using Visual Studio

1. In **Solution Explorer**, open the shortcut menu for the project, and then choose **Properties**.
2. Under the **Build** tab you'll find a **Strong naming** node.
3. Select the **Sign the assembly** checkbox, which expands the options.
4. Select the **Browse** button to choose a **Strong name key file** path.

ⓘ Note

In order to **delay sign** an assembly, choose a public key file.



Create and sign an assembly with a strong name by using the Assembly Linker

Open [Visual Studio Developer Command Prompt](#) or [Visual Studio Developer PowerShell](#), and enter the following command:

```
al /out:<assemblyName> <moduleName> /keyfile:<keyfileName>
```

Where:

- *assemblyName* is the name of the strongly signed assembly (a *.dll* or *.exe* file) that Assembly Linker will emit.
- *moduleName* is the name of a .NET Framework code module (a *.netmodule* file) that includes one or more types. You can create a *.netmodule* file by compiling your code with the `/target:module` switch in C# or Visual Basic.
- *keyfileName* is the name of the container or file that contains the key pair. Assembly Linker interprets a relative path in relation to the current directory.

The following example signs the assembly *MyAssembly.dll* with a strong name by using the key file *sgKey.snk*.

```
Console  
al /out:MyAssembly.dll MyModule.netmodule /keyfile:sgKey.snk
```

For more information about this tool, see [Assembly Linker](#).

Sign an assembly with a strong name by using attributes

1. Add the [System.Reflection.AssemblyKeyFileAttribute](#) or [AssemblyKeyNameAttribute](#) attribute to your source code file, and specify the name of the file or container that contains the key pair to use when signing the assembly with a strong name.

2. Compile the source code file normally.

 **Note**

The C# and Visual Basic compilers issue compiler warnings (CS1699 and BC41008, respectively) when they encounter the [AssemblyKeyFileAttribute](#) or [AssemblyKeyNameAttribute](#) attribute in source code. You can ignore the warnings.

The following example uses the [AssemblyKeyFileAttribute](#) attribute with a key file called *keyfile.snk*, which is located in the directory where the assembly is compiled.

C#

```
[assembly:AssemblyKeyFileAttribute("keyfile.snk")]
```

You can also delay sign an assembly when compiling your source file. For more information, see [Delay-sign an assembly](#).

Sign an assembly with a strong name by using the compiler

Compile your source code file or files with the `/keyfile` or `/delaysign` compiler option in C# and Visual Basic, or the `/KEYFILE` or `/DELAYSIGN` linker option in C++. After the option name, add a colon and the name of the key file. When using command-line compilers, you can copy the key file to the directory that contains your source code files.

For information on delay signing, see [Delay-sign an assembly](#).

The following example uses the C# compiler and signs the assembly *UtilityLibrary.dll* with a strong name by using the key file *sgKey.snk*.

Windows Command Prompt

```
csc /t:library UtilityLibrary.cs /keyfile:sgKey.snk
```

See also

- [Create and use strong-named assemblies](#)
- [How to: Create a public-private key pair](#)
- [Al.exe \(Assembly Linker\)](#)
- [Delay-sign an assembly](#)
- [Strong-name APIs throw PlatformNotSupportedException](#)
- [Manage assembly and manifest signing](#)
- [Signing page, Project Designer](#)

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

Enhanced strong naming

Article • 08/23/2022

A strong name signature is an identity mechanism in the .NET Framework for identifying assemblies. It is a public-key digital signature that is typically used to verify the integrity of data being passed from an originator (signer) to a recipient (verifier). This signature is used as a unique identity for an assembly and ensures that references to the assembly are not ambiguous. The assembly is signed as part of the build process and then verified when it is loaded.

Strong name signatures help prevent malicious parties from tampering with an assembly and then re-signing the assembly with the original signer's key. However, strong name keys don't contain any reliable information about the publisher, nor do they contain a certificate hierarchy. A strong name signature does not guarantee the trustworthiness of the person who signed the assembly or indicate whether that person was a legitimate owner of the key; it indicates only that the owner of the key signed the assembly. Therefore, we do not recommend using a strong name signature as a security validator for trusting third-party code. Microsoft Authenticode is the recommended way to authenticate code.

Limitations of conventional strong names

The strong naming technology used in versions before the .NET Framework 4.5 has the following shortcomings:

- Keys are constantly under attack, and improved techniques and hardware make it easier to infer a private key from a public key. To guard against attacks, larger keys are necessary. .NET Framework versions before the .NET Framework 4.5 provide the ability to sign with any size key (the default size is 1024 bits), but signing an assembly with a new key breaks all binaries that reference the older identity of the assembly. Therefore, it is extremely difficult to upgrade the size of a signing key if you want to maintain compatibility.
- Strong name signing supports only the SHA-1 algorithm. SHA-1 has recently been found to be inadequate for secure hashing applications. Therefore, a stronger algorithm (SHA-256 or greater) is necessary. It is possible that SHA-1 will lose its FIPS-compliant standing, which would present problems for those who choose to use only FIPS-compliant software and algorithms.

Advantages of enhanced strong names

The main advantages of enhanced strong names are compatibility with pre-existing strong names and the ability to claim that one identity is equivalent to another:

- Developers who have pre-existing signed assemblies can migrate their identities to the SHA-2 algorithms while maintaining compatibility with assemblies that reference the old identities.
- Developers who create new assemblies and are not concerned with pre-existing strong name signatures can use the more secure SHA-2 algorithms and sign the assemblies as they always have.

Use enhanced strong names

Strong name keys consist of a signature key and an identity key. The assembly is signed with the signature key and is identified by the identity key. Prior to .NET Framework 4.5, these two keys were identical. Starting with .NET Framework 4.5, the identity key remains the same as in earlier .NET Framework versions, but the signature key is enhanced with a stronger hash algorithm. In addition, the signature key is signed with the identity key to create a counter-signature.

The [AssemblySignatureKeyAttribute](#) attribute enables the assembly metadata to use the pre-existing public key for assembly identity, which allows old assembly references to continue to work. The [AssemblySignatureKeyAttribute](#) attribute uses the counter-signature to ensure that the owner of the new signature key is also the owner of the old identity key.

Sign with SHA-2, without key migration

Run the following commands from a command prompt to sign an assembly without migrating a strong name signature:

1. Generate the new identity key (if necessary).

```
Console  
sn -k IdentityKey.snk
```

2. Extract the identity public key, and specify that a SHA-2 algorithm should be used when signing with this key.

```
Console
```

```
sn -p IdentityKey.snk IdentityPubKey.snk sha256
```

3. Delay-sign the assembly with the identity public key file.

Console

```
csc MyAssembly.cs /keyfile:IdentityPubKey.snk /delaySign+
```

4. Re-sign the assembly with the full identity key pair.

Console

```
sn -Ra MyAssembly.exe IdentityKey.snk
```

Sign with SHA-2, with key migration

Run the following commands from a command prompt to sign an assembly with a migrated strong name signature.

1. Generate an identity and signature key pair (if necessary).

Console

```
sn -k IdentityKey.snk  
sn -k SignatureKey.snk
```

2. Extract the signature public key, and specify that a SHA-2 algorithm should be used when signing with this key.

Console

```
sn -p SignatureKey.snk SignaturePubKey.snk sha256
```

3. Extract the identity public key, which determines the hash algorithm that generates a counter-signature.

Console

```
sn -p IdentityKey.snk IdentityPubKey.snk
```

4. Generate the parameters for an [AssemblySignatureKeyAttribute](#) attribute, and attach the attribute to the assembly.

Console

```
sn -a IdentityPubKey.snk IdentityKey.snk SignaturePubKey.snk
```

This produces output similar to the following.

Output

```
Information for key migration attribute.  
(System.Reflection.AssemblySignatureKeyAttribute):  
publicKey=  
002400000c8000094000000602000002400052534131004000010001005a3a81  
ac0a519  
d96244a9c589fc147c7d403e40ccf184fc290bdd06c7339389a76b738e255a2bce1d56c  
3e7e936  
e4fc87d45adc82ca94c716b50a65d39d373eea033919a613e4341c66863cb2dc622bcb5  
41762b4  
3893434d219d1c43f07e9c83fada2aed400b9f6e44ff05e3ecde6c2827830b8f43f7ac8  
e3270a3  
4d153cdd  
  
counterSignature=  
e3cf7c211678c4d1a7b8fb20276c894ab74c29f0b5a34de4d61e63d4a997222f78cdcbf  
e4c91eb  
e1ddf9f3505a32edcb2a76f34df0450c4f61e376b70fa3cdeb7374b1b8e2078b121e2ee  
6e8c6a8  
ed661cc35621b4af53ac29c9e41738f199a81240e8fd478c887d1a30729d34e954a97cd  
dce66e3  
ae5fec2c682e57b7442738
```

This output can then be transformed into an AssemblySignatureKeyAttribute.

C#

```
[assembly:System.Reflection.AssemblySignatureKeyAttribute(  
"002400000c8000094000000602000002400052534131004000010001005a3a8  
1ac0a519d96244a9c589fc147c7d403e40ccf184fc290bdd06c7339389a76b738e255a2  
bce1d56c3e7e936e4fc87d45adc82ca94c716b50a65d39d373eea033919a613e4341c66  
863cb2dc622bcb541762b43893434d219d1c43f07e9c83fada2aed400b9f6e44ff05e3e  
cde6c2827830b8f43f7ac8e3270a34d153cdd",  
"e3cf7c211678c4d1a7b8fb20276c894ab74c29f0b5a34de4d61e63d4a997222f78cdcb  
fe4c91eb1ddf9f3505a32edcb2a76f34df0450c4f61e376b70fa3cdeb7374b1b8e2078  
b121e2ee6e8c6a8ed661cc35621b4af53ac29c9e41738f199a81240e8fd478c887d1a30  
729d34e954a97cddce66e3ae5fec2c682e57b7442738"  
)]
```

5. Delay-sign the assembly with the identity public key.

Console

```
csc MyAssembly.cs /keyfile:IdentityPubKey.snk /delaySign+
```

6. Fully sign the assembly with the signature key pair.

Console

```
sn -Ra MyAssembly.exe SignatureKey.snk
```

See also

- [Create and use strong-named assemblies](#)

 [Collaborate with us on GitHub](#)

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

How to: Reference a strong-named assembly

Article • 09/15/2021

The process for referencing types or resources in a strong-named assembly is usually transparent. You can make the reference either at compile time (early binding) or at run time.

A compile-time reference occurs when you indicate to the compiler that the assembly to be compiled explicitly references another assembly. When you use compile-time referencing, the compiler automatically gets the public key of the targeted strong-named assembly and places it in the assembly reference of the assembly being compiled.

ⓘ Note

A strong-named assembly can only use types from other strong-named assemblies. Otherwise, the security of the strong-named assembly would be compromised.

Make a compile-time reference to a strong-named assembly

At a command prompt, type the following command:

`<compiler command> /reference:<assembly name>`

In this command, *compiler command* is the compiler command for the language you are using, and *assembly name* is the name of the strong-named assembly being referenced. You can also use other compiler options, such as the `/t:library` option for creating a library assembly.

The following example creates an assembly called *myAssembly.dll* that references a strong-named assembly called *myLibAssembly.dll* from a code module called *myAssembly.cs*.

Windows Command Prompt

```
csc /t:library myAssembly.cs /reference:myLibAssembly.dll
```

Make a run-time reference to a strong-named assembly

When you make a run-time reference to a strong-named assembly, for example by using the [Assembly.Load](#) or [Assembly.GetType](#) method, you must use the display name of the referenced strong-named assembly. The syntax of a display name is as follows:

<assembly name>, <version number>, <culture>, <public key token>

For example:

Console

```
myDll, Version=1.1.0.0, Culture=en, PublicKeyToken=03689116d3a4ae33
```

In this example, `PublicKeyToken` is the hexadecimal form of the public key token. If there is no culture value, use `Culture=neutral`.

The following code example shows how to use this information with the [Assembly.Load](#) method.

C#

```
Assembly myDll =  
    Assembly.Load("myDll, Version=1.0.0.1, Culture=neutral,  
    PublicKeyToken=9b35aa32c18d4fb1");
```

You can print the hexadecimal format of the public key and public key token for a specific assembly by using the following [Strong Name \(Sn.exe\)](#) command:

`sn -Tp <assembly>`

If you have a public key file, you can use the following command instead (note the difference in case on the command-line option):

`sn -tp <public key file>`

See also

- [Create and use strong-named assemblies](#)

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

How to: Disable the strong-name bypass feature

Article • 09/15/2021

Starting with the .NET Framework version 3.5 Service Pack 1 (SP1), strong-name signatures are not validated when an assembly is loaded into a full-trust [AppDomain](#) object, such as the default [AppDomain](#) for the `MyComputer` zone. This is referred to as the strong-name bypass feature. In a full-trust environment, demands for [StrongNameIdentityPermission](#) always succeed for signed, full-trust assemblies regardless of their signature. The only restriction is that the assembly must be fully trusted because its zone is fully trusted. Because the strong name is not a determining factor under these conditions, there is no reason for it to be validated. Bypassing the validation of strong-name signatures provides significant performance improvements.

The bypass feature applies to any full-trust assembly that is not delay-signed and that is loaded into any full-trust [AppDomain](#) from the directory specified by its [ApplicationBase](#) property.

You can override the bypass feature for all applications on a computer by setting a registry key value. You can override the setting for a single application by using an application configuration file. You cannot reinstate the bypass feature for a single application if it has been disabled by the registry key.

When you override the bypass feature, the strong name is validated only for correctness; it is not checked for a [StrongNameIdentityPermission](#). If you want to confirm a specific strong name, you have to perform that check separately.

Important

The ability to force strong-name validation depends on a registry key, as described in the following procedure. If an application is running under an account that does not have access control list (ACL) permission to access that registry key, the setting is ineffective. You must ensure that ACL rights are configured for this key so that it can be read for all assemblies.

Disable the strong-name bypass feature for all applications

- On 32-bit computers, in the system registry, create a DWORD entry with a value of 0 named `AllowStrongNameBypass` under the `HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\.NETFramework` key.
- On 64-bit computers, in the system registry, create a DWORD entry with a value of 0 named `AllowStrongNameBypass` under the `HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\.NETFramework` and `HKEY_LOCAL_MACHINE\SOFTWARE\Wow6432Node\Microsoft\.NETFramework` keys.

Disable the strong-name bypass feature for a single application

1. Open or create the application configuration file.

For more information about this file, see the Application Configuration Files section in [Configure apps](#).

2. Add the following entry:

XML

```
<configuration>
  <runtime>
    <bypassTrustedAppStrongNames enabled="false" />
  </runtime>
</configuration>
```

You can restore the bypass feature for the application by removing the configuration file setting or by setting the attribute to `true`.

ⓘ Note

You can turn strong-name validation on and off for an application only if the bypass feature is enabled for the computer. If the bypass feature has been turned off for the computer, strong names are validated for all applications and you cannot bypass validation for a single application.

See also

- [Sn.exe \(Strong Name Tool\)](#)

- <bypassTrustedAppStrongNames> element
- Create and use strong-named assemblies

 Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

Delay-sign an assembly

Article • 04/29/2022

An organization can have a closely guarded key pair that developers can't access on a daily basis. The public key is often available, but access to the private key is restricted to only a few individuals. When developing assemblies with strong names, each assembly that references the strong-named target assembly contains the token of the public key used to give the target assembly a strong name. This requires that the public key be available during the development process.

You can use delayed or partial signing at build time to reserve space in the portable executable (PE) file for the strong name signature, but defer the actual signing until some later stage, usually just before shipping the assembly.

To delay-sign an assembly:

1. Get the public key portion of the key pair from the organization that will do the eventual signing. Typically this key is in the form of an `.snk` file, which can be created using the [Strong Name tool \(Sn.exe\)](#) provided by the Windows SDK.
2. Annotate the source code for the assembly with two custom attributes from [System.Reflection](#):
 - [AssemblyKeyFileAttribute](#), which passes the name of the file containing the public key as a parameter to its constructor.
 - [AssemblyDelaySignAttribute](#), which indicates that delay signing is being used by passing `true` as a parameter to its constructor.

For example:

C#

```
[assembly:AssemblyKeyFileAttribute("myKey.snk")]
[assembly:AssemblyDelaySignAttribute(true)]
```

3. The compiler inserts the public key into the assembly manifest and reserves space in the PE file for the full strong name signature. The real public key must be stored while the assembly is built so that other assemblies that reference this assembly can obtain the key to store in their own assembly reference.
4. Because the assembly does not have a valid strong name signature, the verification of that signature must be turned off. You can do this by using the `-Vr` option with

the Strong Name tool.

The following example turns off verification for an assembly called *myAssembly.dll*.

Console

```
sn -Vr myAssembly.dll
```

To turn off verification on platforms where you can't run the Strong Name tool, such as Advanced RISC Machine (ARM) microprocessors, use the **-Vk** option to create a registry file. Import the registry file into the registry on the computer where you want to turn off verification. The following example creates a registry file for *myAssembly.dll*.

Console

```
sn -Vk myRegFile.reg myAssembly.dll
```

With either the **-Vr** or **-Vk** option, you can optionally include an *.snk* file for test key signing.

⚠ Warning

Do not rely on strong names for security. They provide a unique identity only.

ⓘ Note

If you use delay signing during development with Visual Studio on a 64-bit computer, and you compile an assembly for **Any CPU**, you might have to apply the **-Vr** option twice. (In Visual Studio, **Any CPU** is a value of the **Platform Target** build property; when you compile from the command line, it is the default.) To run your application from the command line or from File Explorer, use the 64-bit version of the [Sn.exe \(Strong Name tool\)](#) to apply the **-Vr** option to the assembly. To load the assembly into Visual Studio at design time (for example, if the assembly contains components that are used by other assemblies in your application), use the 32-bit version of the strong-name tool. This is because the just-in-time (JIT) compiler compiles the assembly to 64-bit native code when the assembly is run from the command line, and to 32-bit native code when the assembly is loaded into the design-time environment.

5. Later, usually just before shipping, you submit the assembly to your organization's signing authority for the actual strong name signing using the `-R` option with the Strong Name tool.

The following example signs an assembly called `myAssembly.dll` with a strong name using the `sgKey.snk` key pair.

Console

```
sn -R myAssembly.dll sgKey.snk
```

See also

- [Create assemblies](#)
- [How to: Create a public-private key pair](#)
- [Sn.exe \(Strong Name tool\)](#)

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

How to: View assembly contents

Article • 01/12/2022

You can use the [Illdasm.exe \(IL Disassembler\)](#) to view Microsoft intermediate language (MSIL) information in a file. If the file being examined is an assembly, this information can include the assembly's attributes and references to other modules and assemblies. This information can be helpful in determining whether a file is an assembly or part of an assembly and whether the file has references to other modules or assemblies.

To display the contents of an assembly using *Illdasm.exe*, enter **ildasm <assembly name>** at a command prompt. For example, the following command disassembles the *Hello.exe* assembly.

Windows Command Prompt

```
ildasm Hello.exe
```

To view assembly manifest information, double-click the **Manifest** icon in the MSIL Disassembler window.

Example

The following example starts with a basic "Hello World" program. After compiling the program, use *Illdasm.exe* to disassemble the *Hello.exe* assembly and view the assembly manifest.

C#

```
using System;

class MainApp
{
    public static void Main()
    {
        Console.WriteLine("Hello World using C#!");
    }
}
```

Running the command *ildasm.exe* on the *Hello.exe* assembly and double-clicking the **Manifest** icon in the MSIL Disassembler window produces the following output:

Output

```

// Metadata version: v4.0.30319
.assembly extern mscorel
{
    .publickeytoken = (B7 7A 5C 56 19 34 E0 89 ) //z\V.4..
    .ver 4:0:0:0
}
.assembly Hello
{
    .custom instance void
[mscorel]System.Runtime.CompilerServices.CompilationRelaxationsAttribute::ctor(int32) = ( 01 00 08 00 00 00 00 00 )
    .custom instance void
[mscorel]System.Runtime.CompilerServices.RuntimeCompatibilityAttribute::ctor() = ( 01 00 01 00 54 02 16 57 72 61 70 4E 6F 6E 45 78 //....T..WrapNonEx

63 65 70 74 69 6F 6E 54 68 72 6F 77 73 01 ) // captionThrows.
    .hash algorithm 0x00008004
    .ver 0:0:0:0
}
.module Hello.exe
// MVID: {7C2770DB-1594-438D-BAE5-98764C39CCCA}
.imagebase 0x00400000
.file alignment 0x00000200
.stackreserve 0x00100000
.subsystem 0x0003 // WINDOWS_CUI
.corflags 0x00000001 // ILOONLY
// Image base: 0x00600000

```

The following table describes each directive in the assembly manifest of the *Hello.exe* assembly used in the example:

| Directive | Description |
|-------------------------------------|--|
| .assembly extern <assembly name> | Specifies another assembly that contains items referenced by the current module (in this example, <code>mscorlib</code>). |
| .publickeytoken <token> | Specifies the token of the actual key of the referenced assembly. |
| .ver <version number> | Specifies the version number of the referenced assembly. |
| .assembly <assembly name> | Specifies the assembly name. |
| .hash algorithm <int32 value> | Specifies the hash algorithm used. |
| .ver <version number> | Specifies the version number of the assembly. |

| Directive | Description |
|---------------------|--|
| .module <file name> | Specifies the name of the modules that make up the assembly. In this example, the assembly consists of only one file. |
| .subsystem <value> | Specifies the application environment required for the program. In this example, the value 3 indicates that this executable is run from a console. |
| .corflags | Currently a reserved field in the metadata. |

An assembly manifest can contain a number of different directives, depending on the contents of the assembly. For an extensive list of the directives in the assembly manifest, see the Ecma documentation, especially "Partition II: Metadata Definition and Semantics" and "Partition III: CIL Instruction Set":

- [ECMA C# and Common Language Infrastructure standards](#)
- [Standard ECMA-335 - Common Language Infrastructure \(CLI\)](#) ↗

See also

- [Application domains and assemblies](#)
- [Application domains and assemblies how-to topics](#)
- [Ildasm.exe \(IL Disassembler\)](#)

 Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

Type forwarding in the common language runtime

Article • 10/14/2022

Type forwarding allows you to move a type to another assembly without having to recompile applications that use the original assembly.

For example, suppose an application uses the `Example` class in an assembly named `Utility.dll`. The developers of `Utility.dll` might decide to refactor the assembly, and in the process they might move the `Example` class to another assembly. If the old version of `Utility.dll` is replaced by the new version of `Utility.dll` and its companion assembly, the application that uses the `Example` class fails because it cannot locate the `Example` class in the new version of `Utility.dll`.

The developers of `Utility.dll` can avoid this by forwarding requests for the `Example` class, using the `TypeForwardedToAttribute` attribute. If the attribute has been applied to the new version of `Utility.dll`, requests for the `Example` class are forwarded to the assembly that now contains the class. The existing application continues to function normally, without recompilation.

Forward a type

There are four steps to forwarding a type:

1. Move the source code for the type from the original assembly to the destination assembly.
2. In the assembly where the type used to be located, add a `TypeForwardedToAttribute` for the type that was moved. The following code shows the attribute for a type named `Example` that was moved.

C#

```
[assembly:TypeForwardedToAttribute(typeof(Example))]
```

3. Compile the assembly that now contains the type.
4. Recompile the assembly where the type used to be located, with a reference to the assembly that now contains the type. For example, if you are compiling a C# file from the command line, use the [References \(C# compiler options\)](#) option to

specify the assembly that contains the type. In C++, use the `#using` directive in the source file to specify the assembly that contains the type.

C# type forwarding example

Continuing from the contrived example description above, imagine you're developing the *Utility.dll*, and you have an `Example` class. The *Utility.csproj* is a basic class library:

```
XML

<Project Sdk="Microsoft.NET.Sdk">

    <PropertyGroup>
        <TargetFramework>net6.0</TargetFramework>
        <Nullable>enable</Nullable>
        <ImplicitUsing>true</ImplicitUsing>
    </PropertyGroup>

</Project>
```

The `Example` class provides a few properties and overrides `Object.ToString`:

```
C#

using System;

namespace Common.Objects;

public class Example
{
    public string Message { get; init; } = "Hi friends!";

    public Guid Id { get; init; } = Guid.NewGuid();

    public DateOnly Date { get; init; } =
DateOnly.FromDateTime(DateTime.Today);

    public sealed override string ToString() =>
        $"[{Id} - {Date}]: {Message}";
}
```

Now, imagine that there is a consuming project and it's represented in the *Consumer* assembly. This consuming project references the *Utility* assembly. As an example, it instantiates the `Example` object and writes it to the console in its *Program.cs* file:

```
C#
```

```
using System;
using Common.Objects;

Example example = new();

Console.WriteLine(example);
```

When the consuming app runs, it will output the state of the `Example` object. At this point, there is no type forwarding as the `Consuming.csproj` references the `Utility.csproj`. However, the developer's of the `Utility` assembly decide to remove the `Example` object as part of a refactoring. This type is moved to a newly created `Common.csproj`.

By removing this type from the `Utility` assembly, the developers are introducing a breaking change. All consuming projects will break when they update to the latest `Utility` assembly.

Instead of requiring the consuming projects to add a new reference to the `Common` assembly, you can forward the type. Since this type was removed from the `Utility` assembly, you'll need to have the `Utility.csproj` reference the `Common.csproj`:

XML

```
<Project Sdk="Microsoft.NET.Sdk">

    <PropertyGroup>
        <TargetFramework>net6.0</TargetFramework>
        <Nullable>enable</Nullable>
        <ImplicitUsing>true</ImplicitUsing>
    </PropertyGroup>

    <ItemGroup>
        <ProjectReference Include=".\\Common\\Common.csproj" />
    </ItemGroup>

</Project>
```

The preceding C# project now references the newly created `Common` assembly. This could be either a `PackageReference` or a `ProjectReference`. The `Utility` assembly needs to provide the type forwarding information. By convention type forward declarations are usually encapsulated in a single file named `TypeForwarders`, consider the following `TypeForwarders.cs` C# file in the `Utility` assembly:

C#

```
using System.Runtime.CompilerServices;
using Common.Objects;
```

```
[assembly:TypeForwardedTo(typeof(Example))]
```

The *Utility* assembly references the *Common* assembly, and it forwards the `Example` type. If you're to compile the *Utility* assembly with the type forwarding declarations and drop the *Utility.dll* into the *Consuming* bin, the consuming app will work without being compiled.

See also

- [TypeForwardedToAttribute](#)
- [Type forwarding \(C++/CLI\)](#)
- [#using directive](#)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

Friend assemblies

Article • 09/15/2021

A *friend assembly* is an assembly that can access another assembly's `internal` (C#) or `Friend` (Visual Basic) types and members. If you add an assembly attribute to *AssemblyA* to identify *AssemblyB* as a friend assembly, you no longer have to mark types and members in *AssemblyA* as public in order for them to be accessed by *AssemblyB*. This is especially convenient in the following scenarios:

- During unit testing, when test code runs in a separate assembly but requires access to members in the assembly being tested that are marked as `internal` in C# or `Friend` in Visual Basic.
- When you are developing a class library and additions to the library are contained in separate assemblies but require access to members in existing assemblies that are marked as `internal` in C# or `Friend` in Visual Basic.

Remarks

You can use the `InternalsVisibleToAttribute` attribute to identify one or more friend assemblies for a given assembly. The following example uses the `InternalsVisibleToAttribute` attribute in *AssemblyA* and specifies assembly *AssemblyB* as a friend assembly. This gives assembly *AssemblyB* access to all types and members in *Assembly A* that are marked as `internal` in C# or `Friend` in Visual Basic.

ⓘ Note

When you compile an assembly like *AssemblyB* that will access internal types or internal members of another assembly like *AssemblyA*, you must explicitly specify the name of the output file (`.exe` or `.dll`) by using the `-out` compiler option. This is required because the compiler has not yet generated the name for the assembly it is building at the time it is binding to external references. For more information, see [OutputAssembly \(C#\)](#) or [-out \(Visual Basic\)](#).

C#

```
using System.Runtime.CompilerServices;
using System;

[assembly: InternalsVisibleTo("AssemblyB")]
```

```

// The class is internal by default.
class FriendClass
{
    public void Test()
    {
        Console.WriteLine("Sample Class");
    }
}

// Public class that has an internal method.
public class ClassWithFriendMethod
{
    internal void Test()
    {
        Console.WriteLine("Sample Method");
    }
}

```

Only assemblies that you explicitly specify as friends can access `internal` (C#) or `Friend` (Visual Basic) types and members. For example, if *AssemblyB* is a friend of *Assembly A* and *Assembly C* references *AssemblyB*, *Assembly C* does not have access to `internal` (C#) or `Friend` (Visual Basic) types in *Assembly A*.

The compiler performs some basic validation of the friend assembly name passed to the [InternalsVisibleToAttribute](#) attribute. If *Assembly A* declares *AssemblyB* as a friend assembly, the validation rules are as follows:

- If *Assembly A* is strong named, *AssemblyB* must also be strong named. The friend assembly name that is passed to the attribute must consist of the assembly name and the public key of the strong-name key that is used to sign *AssemblyB*.

The friend assembly name that is passed to the [InternalsVisibleToAttribute](#) attribute cannot be the strong name of *AssemblyB*. Don't include the assembly version, culture, architecture, or public key token.

- If *Assembly A* is not strong named, the friend assembly name should consist of only the assembly name. For more information, see [How to: Create unsigned friend assemblies](#).
- If *AssemblyB* is strong named, you must specify the strong-name key for *AssemblyB* by using the project setting or the command-line `/keyfile` compiler option. For more information, see [How to: Create signed friend assemblies](#).

The [StrongNameIdentityPermission](#) class also provides the ability to share types, with the following differences:

- [StrongNameIdentityPermission](#) applies to an individual type, while a friend assembly applies to the whole assembly.
- If there are hundreds of types in *Assembly A* that you want to share with *AssemblyB*, you have to add [StrongNameIdentityPermission](#) to all of them. If you use a friend assembly, you only need to declare the friend relationship once.
- If you use [StrongNameIdentityPermission](#), the types you want to share have to be declared as public. If you use a friend assembly, the shared types are declared as `internal` (C#) or `Friend` (Visual Basic).

For information about how to access an assembly's `internal` (C#) or `Friend` (Visual Basic) types and methods from a module file (a file with the `.netmodule` extension), see [ModuleAssemblyName](#) (C#) or [-moduleassemblyname](#) (Visual Basic).

See also

- [InternalsVisibleToAttribute](#)
- [StrongNameIdentityPermission](#)
- [How to: Create unsigned friend assemblies](#)
- [How to: Create signed friend assemblies](#)
- [Assemblies in .NET](#)
- [C# programming guide](#)
- [Programming concepts \(Visual Basic\)](#)

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

How to: Create unsigned friend assemblies

Article • 09/15/2021

This example shows how to use friend assemblies with assemblies that are unsigned.

Create an assembly and a friend assembly

1. Open a command prompt.
2. Create a C# or Visual Basic file named *friend_unsigned_A* that contains the following code. The code uses the `InternalsVisibleToAttribute` attribute to declare *friend_unsigned_B* as a friend assembly.

```
C#  
  
// friend_unsigned_A.cs  
// Compile with:  
// csc /target:library friend_unsigned_A.cs  
using System.Runtime.CompilerServices;  
using System;  
  
[assembly: InternalsVisibleTo("friend_unsigned_B")]  
  
// Type is internal by default.  
class Class1  
{  
    public void Test()  
    {  
        Console.WriteLine("Class1.Test");  
    }  
}  
  
// Public type with internal member.  
public class Class2  
{  
    internal void Test()  
    {  
        Console.WriteLine("Class2.Test");  
    }  
}
```

3. Compile and sign *friend_unsigned_A* by using the following command:

```
C#
```

```
csc /target:library friend_unsigned_A.cs
```

4. Create a C# or Visual Basic file named *friend_unsigned_B* that contains the following code. Because *friend_unsigned_A* specifies *friend_unsigned_B* as a friend assembly, the code in *friend_unsigned_B* can access `internal` (C#) or `Friend` (Visual Basic) types and members from *friend_unsigned_A*.

C#

```
// friend_unsigned_B.cs
// Compile with:
// csc /r:friend_unsigned_A.dll /out:friend_unsigned_B.exe
friend_unsigned_B.cs
public class Program
{
    static void Main()
    {
        // Access an internal type.
        Class1 inst1 = new Class1();
        inst1.Test();

        Class2 inst2 = new Class2();
        // Access an internal member of a public type.
        inst2.Test();

        System.Console.ReadLine();
    }
}
```

5. Compile *friend_unsigned_B* by using the following command.

C#

```
csc /r:friend_unsigned_A.dll /out:friend_unsigned_B.exe
friend_unsigned_B.cs
```

The name of the assembly that is generated by the compiler must match the friend assembly name that is passed to the `InternalsVisibleToAttribute` attribute. You must explicitly specify the name of the output assembly (`.exe` or `.dll`) by using the `-out` compiler option. For more information, see [OutputAssembly \(C# compiler options\)](#) or [-out \(Visual Basic\)](#).

6. Run the *friend_unsigned_B.exe* file.

The program outputs two strings: `Class1.Test` and `Class2.Test`.

.NET security

There are similarities between the [InternalsVisibleToAttribute](#) attribute and the [StrongNameIdentityPermission](#) class. The main difference is that [StrongNameIdentityPermission](#) can demand security permissions to run a particular section of code, whereas the [InternalsVisibleToAttribute](#) attribute controls the visibility of `internal` or `Friend` (Visual Basic) types and members.

See also

- [InternalsVisibleToAttribute](#)
- [Assemblies in .NET](#)
- [Friend assemblies](#)
- [How to: Create signed friend assemblies](#)
- [C# programming guide](#)
- [Programming concepts \(Visual Basic\)](#)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

How to: Create signed friend assemblies

Article • 09/15/2021

This example shows how to use friend assemblies with assemblies that have strong names. Both assemblies must be strong named. Although both assemblies in this example use the same keys, you could use different keys for two assemblies.

Create a signed assembly and a friend assembly

1. Open a command prompt.
2. Use the following sequence of commands with the Strong Name tool to generate a keyfile and to display its public key. For more information, see [Sn.exe \(Strong Name tool\)](#).
 - a. Generate a strong-name key for this example and store it in the file *FriendAssemblies.snk*:

```
sn -k FriendAssemblies.snk
```

- b. Extract the public key from *FriendAssemblies.snk* and put it into *FriendAssemblies.publickey*:

```
sn -p FriendAssemblies.snk FriendAssemblies.publickey
```

- c. Display the public key stored in the file *FriendAssemblies.publickey*:

```
sn -tp FriendAssemblies.publickey
```

3. Create a C# or Visual Basic file named *friend_signed_A* that contains the following code. The code uses the [InternalsVisibleToAttribute](#) attribute to declare *friend_signed_B* as a friend assembly.

The Strong Name tool generates a new public key every time it runs. Therefore, you must replace the public key in the following code with the public key you just generated, as shown in the following example.

C#

```
// friend_signed_A.cs
// Compile with:
// csc /target:library /keyfile:FriendAssemblies.snk friend_signed_A.cs
```

```
using System.Runtime.CompilerServices;

[assembly: InternalsVisibleTo("friend_signed_B,
PublicKey=002400000480000940000006020000024000052534131000400001000
100e3aedce99b7e10823920206f8e46cd5558b4ec7345bd1a5b201ffe71660625dcb8f9
a08687d881c8f65a0dcf042f81475d2e88f3e3e273c8311ee40f952db306c02fbfc5d8b
c6ee1e924e6ec8fe8c01932e0648a0d3e5695134af3bb7fab370d3012d083fa6b83179d
d3d031053f72fc1f7da8459140b0af5afc4d2804deccb6")]
class Class1
{
    public void Test()
    {
        System.Console.WriteLine("Class1.Test");
        System.Console.ReadLine();
    }
}
```

4. Compile and sign *friend_signed_A* by using the following command.

```
C#  
  
csc /target:library /keyfile:FriendAssemblies.snk friend_signed_A.cs
```

5. Create a C# or Visual Basic file named *friend_signed_B* that contains the following code. Because *friend_signed_A* specifies *friend_signed_B* as a friend assembly, the code in *friend_signed_B* can access `internal` (C#) or `Friend` (Visual Basic) types and members from *friend_signed_A*. The file contains the following code.

```
C#  
  
// friend_signed_B.cs
// Compile with:
// csc /keyfile:FriendAssemblies.snk /r:friend_signed_A.dll
// /out:friend_signed_B.exe friend_signed_B.cs
public class Program
{
    static void Main()
    {
        Class1 inst = new Class1();
        inst.Test();
    }
}
```

6. Compile and sign *friend_signed_B* by using the following command.

```
C#  
  
csc /keyfile:FriendAssemblies.snk /r:friend_signed_A.dll
```

```
/out:friend_signed_B.exe friend_signed_B.cs
```

The name of the assembly generated by the compiler must match the friend assembly name passed to the [InternalsVisibleToAttribute](#) attribute. You must explicitly specify the name of the output assembly (.exe or .dll) by using the `-out` compiler option. For more information, see [OutputAssembly \(C# compiler options\)](#) or [-out \(Visual Basic\)](#).

7. Run the `friend_signed_B.exe` file.

The program outputs the string `Class1.Test`.

.NET security

There are similarities between the [InternalsVisibleToAttribute](#) attribute and the [StrongNameIdentityPermission](#) class. The main difference is that [StrongNameIdentityPermission](#) can demand security permissions to run a particular section of code, whereas the [InternalsVisibleToAttribute](#) attribute controls the visibility of `internal` (C#) or `Friend` (Visual Basic) types and members.

See also

- [InternalsVisibleToAttribute](#)
- [Assemblies in .NET](#)
- [Friend assemblies](#)
- [How to: Create unsigned friend assemblies](#)
- [KeyFile \(C#\)](#)
- [-keyfile \(Visual Basic\)](#)
- [Sn.exe \(Strong Name tool\)](#)
- [Create and use strong-named assemblies](#)
- [C# programming guide](#)
- [Programming concepts \(Visual Basic\)](#)

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

more information, see [our contributor guide](#).

 [Provide product feedback](#)

How to: Determine if a file is an assembly

Article • 12/21/2022

A file is an assembly if and only if it is managed, and contains an assembly entry in its metadata. For more information on assemblies and metadata, see [Assembly manifest](#).

How to manually determine if a file is an assembly

1. Start the [Ilasm.exe \(IL Disassembler\)](#).
2. Load the file you want to test.
3. If ILDASM reports that the file is not a portable executable (PE) file, then it is not an assembly. For more information, see the topic [How to: View assembly contents](#).

How to programmatically determine if a file is an assembly

Using the AssemblyName class

1. Call the [AssemblyName.GetAssemblyName](#) method, passing the full file path and name of the file you are testing.
2. If a [BadImageFormatException](#) exception is thrown, the file is not an assembly.

This example tests a DLL to see if it is an assembly.

C#

```
using System;
using System.IO;
using System.Reflection;
using System.Runtime.InteropServices;

static class ExampleAssemblyName
{
    public static void CheckAssembly()
    {
        try
        {
```

```

        string path = Path.Combine(
            RuntimeEnvironment.GetRuntimeDirectory(),
            "System.Net.dll");

        AssemblyName testAssembly = AssemblyName.GetAssemblyName(path);
        Console.WriteLine("Yes, the file is an assembly.");
    }
    catch (FileNotFoundException)
    {
        Console.WriteLine("The file cannot be found.");
    }
    catch (BadImageFormatException)
    {
        Console.WriteLine("The file is not an assembly.");
    }
    catch (FileLoadException)
    {
        Console.WriteLine("The assembly has already been loaded.");
    }
}

/* Output:
Yes, the file is an assembly.
*/
}

```

The [GetAssemblyName](#) method loads the test file, and then releases it once the information is read.

Using the PEReader class

1. If you're targeting .NET Standard or .NET Framework, install the [System.Reflection.Metadata](#) NuGet package. (When targeting .NET Core or .NET 5+, this step isn't required because this library is included in the shared framework.)
2. Create a [System.IO.FileStream](#) instance to read data from the file you're testing.
3. Create a [System.Reflection.PortableExecutable.PEReader](#) instance, passing your file stream into the constructor.
4. Check the value of the [HasMetadata](#) property. If the value is `false`, the file is not an assembly.
5. Call the [GetMetadataReader](#) method on the PE reader instance to create a metadata reader.

6. Check the value of the [IsAssembly](#) property. If the value is `true`, the file is an assembly.

Unlike the [GetAssemblyName](#) method, the [PEReader](#) class does not throw an exception on native Portable Executable (PE) files. This enables you to avoid the extra performance cost caused by exceptions when you need to check such files. You still need to handle exceptions in case the file does not exist or is not a PE file.

This example shows how to determine if a file is an assembly using the [PEReader](#) class.

C#

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Reflection.Metadata;
using System.Reflection.PortableExecutable;
using System.Runtime.InteropServices;

static class ExamplePeReader
{
    static bool IsAssembly(string path)
    {
        using var fs = new FileStream(path, FileMode.Open, FileAccess.Read,
FileShare.ReadWrite);

        // Try to read CLI metadata from the PE file.
        using var peReader = new PEReader(fs);

        if (!peReader.HasMetadata)
        {
            return false; // File does not have CLI metadata.
        }

        // Check that file has an assembly manifest.
        MetadataReader reader = peReader.GetMetadataReader();
        return reader.IsAssembly;
    }

    public static void CheckAssembly()
    {
        string path = Path.Combine(
            RuntimeEnvironment.GetRuntimeDirectory(),
            "System.Net.dll");

        try
        {
            if (IsAssembly(path))
            {
                Console.WriteLine("Yes, the file is an assembly.");
            }
            else
        }
```

```
        {
            Console.WriteLine("The file is not an assembly.");
        }
    }
    catch (BadImageFormatException)
    {
        Console.WriteLine("The file is not an executable.");
    }
    catch (FileNotFoundException)
    {
        Console.WriteLine("The file cannot be found.");
    }
}

/* Output:
Yes, the file is an assembly.
*/
}
```

See also

- [AssemblyName](#)
- [C# programming guide](#)
- [Programming concepts \(Visual Basic\)](#)
- [Assemblies in .NET](#)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

Open a documentation issue

Provide product feedback

How to: Load and unload assemblies

Article • 09/15/2021

The assemblies referenced by your program will automatically be loaded by the common language runtime, but it is also possible to dynamically load specific assemblies into the current application domain. For more information, see [How to: Load assemblies into an application domain](#).

In .NET Framework, there is no way to unload an individual assembly without unloading all of the application domains that contain it. Even if the assembly goes out of scope, the actual assembly file will remain loaded until all application domains that contain it are unloaded. In .NET Core, the `System.Runtime.Loader.AssemblyLoadContext` class handles the unloading of assemblies. For more information, see [How to use and debug assembly unloadability in .NET Core](#).

Load and unload assemblies

To load an assembly into an application domain, use one of the several load methods contained in the classes `AppDomain` and `Assembly`. For more information, see [How to: Load assemblies into an application domain](#). Note that .NET Core supports only a single application domain.

To unload an assembly in the .NET Framework, you must unload all of the application domains that contain it. To unload an application domain, use the `AppDomain.Unload` method. For more information, see [How to: Unload an application domain](#).

If you want to unload some assemblies but not others in a .NET Framework application, consider creating a new application domain, executing the code inside that domain, and then unloading that application domain. For more information, see [How to: Unload an application domain](#).

See also

- [C# programming guide](#)
- [Programming concepts \(Visual Basic\)](#)
- [Assemblies in .NET](#)
- [How to: Load assemblies into an application domain](#)

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

Walkthrough: Embed types from managed assemblies in Visual Studio

Article • 10/14/2022

If you embed type information from a strong-named managed assembly, you can loosely couple types in an application to achieve version independence. That is, your program can be written to use types from any version of a managed library without having to be recompiled for each new version.

Type embedding is frequently used with COM interop, such as an application that uses automation objects from Microsoft Office. Embedding type information enables the same build of a program to work with different versions of Microsoft Office on different computers. However, you can also use type embedding with fully managed solutions.

After you specify the public interfaces that can be embedded, you create runtime classes that implement those interfaces. A client program can embed the type information for the interfaces at design time by referencing the assembly that contains the public interfaces and setting the `Embed Interop Types` property of the reference to `True`. The client program can then load instances of the runtime objects typed as those interfaces. This is equivalent to using the command line compiler and referencing the assembly by using the [EmbedInteropTypes compiler option](#).

If you create a new version of your strong-named runtime assembly, the client program doesn't have to be recompiled. The client program continues to use whichever version of the runtime assembly is available to it, using the embedded type information for the public interfaces.

In this walkthrough, you:

1. Create a strong-named assembly with a public interface containing type information that can be embedded.
2. Create a strong-named runtime assembly that implements the public interface.
3. Create a client program that embeds the type information from the public interface and creates an instance of the class from the runtime assembly.
4. Modify and rebuild the runtime assembly.
5. Run the client program to see that it uses the new version of the runtime assembly without having to be recompiled.

ⓘ Note

Your computer might show different names or locations for some of the Visual Studio user interface elements in the following instructions. The Visual Studio edition that you have and the settings that you use determine these elements. For more information, see [Personalizing the IDE](#).

Conditions and limitations

You can embed type information from an assembly under the following conditions:

- The assembly exposes at least one public interface.
- The embedded interfaces are annotated with `ComImport` attributes and `Guid` attributes with unique GUIDs.
- The assembly is annotated with the `ImportedFromTypeLib` attribute or the `PrimaryInteropAssembly` attribute, and an assembly-level `Guid` attribute. The Visual C# and Visual Basic project templates include an assembly-level `Guid` attribute by default.

Because the primary function of type embedding is to support COM interop assemblies, the following limitations apply when you embed type information in a fully-managed solution:

- Only attributes specific to COM interop are embedded. Other attributes are ignored.
- If a type uses generic parameters, and the type of the generic parameter is an embedded type, that type cannot be used across an assembly boundary. Examples of crossing an assembly boundary include calling a method from another assembly or deriving a type from a type defined in another assembly.
- Constants are not embedded.
- The `System.Collections.Generic.Dictionary<TKey,TValue>` class does not support an embedded type as a key. You can implement your own dictionary type to support an embedded type as a key.

Create an interface

The first step is to create the type equivalence interface assembly.

1. In Visual Studio, select **File > New > Project**.
2. In the **Create a new project** dialog box, type *class library* in the **Search for templates** box. Select either the C# or Visual Basic **Class Library (.NET Framework)** template from the list, and then select **Next**.

3. In the **Configure your new project** dialog box, under **Project name**, type `TypeEquivalenceInterface`, and then select **Create**. The new project is created.
4. In **Solution Explorer**, right-click the `Class1.cs` or `Class1.vb` file, select **Rename**, and rename the file from `Class1` to `ISampleInterface`. Respond **Yes** to the prompt to also rename the class to `ISampleInterface`. This class represents the public interface for the class.
5. In **Solution Explorer**, right-click the `TypeEquivalenceInterface` project, and then select **Properties**.
6. Select **Build** on the left pane of the **Properties** screen, and set the **Output path** to a location on your computer, such as `C:\TypeEquivalenceSample`. You use the same location throughout this walkthrough.
7. Select **Build > Strong naming** on the left pane of the **Properties** screen, and then select the **Sign the assembly** check box. In the **Strong name key file**, select **Browse**.
8. Navigate to and select the `key.snk` file you created in the `TypeEquivalenceInterface` project, and then select **OK**. For more information, see [Create a public-private key pair](#).
9. Open the `ISampleInterface` class file in the code editor, and replace its contents with the following code to create the `ISampleInterface` interface:

```
C#  
  
using System;  
using System.Runtime.InteropServices;  
  
namespace TypeEquivalenceInterface  
{  
    [ComImport]  
    [Guid("8DA56996-A151-4136-B474-32784559F6DF")]  
    public interface ISampleInterface  
    {  
        void GetUserInput();  
        string UserInput { get; }  
    }  
}
```

10. On the **Tools** menu, select **Create Guid**, and in the **Create GUID** dialog box, select **Registry Format**. Select **Copy**, and then select **Exit**.

11. In the `Guid` attribute of your code, replace the sample GUID with the GUID you copied, and remove the braces (`{ }`).
12. In **Solution Explorer**, expand the **Properties** folder and select the `AssemblyInfo.cs` or `AssemblyInfo.vb` file. In the code editor, add the following attribute to the file:

```
C#  
[assembly: ImportedFromTypeLib("")]
```

13. Select **File > Save All** or press `Ctrl + Shift + S` to save the files and project.
14. In **Solution Explorer**, right-click the `TypeEquivalenceInterface` project and select **Build**. The class library DLL file is compiled and saved to the specified build output path, for example `C:\TypeEquivalenceSample`.

Create a runtime class

Next, create the type equivalence runtime class.

1. In Visual Studio, select **File > New > Project**.
2. In the **Create a new project** dialog box, type *class library* in the **Search for templates** box. Select either the C# or Visual Basic **Class Library (.NET Framework)** template from the list, and then select **Next**.
3. In the **Configure your new project** dialog box, under **Project name**, type `TypeEquivalenceRuntime`, and then select **Create**. The new project is created.
4. In **Solution Explorer**, right-click the `Class1.cs` or `Class1.vb` file, select **Rename**, and rename the file from `Class1` to `SampleClass`. Respond **Yes** to the prompt to also rename the class to `SampleClass`. This class implements the `ISampleInterface` interface.
5. In **Solution Explorer**, right-click the `TypeEquivalenceInterface` project and select **Properties**.
6. Select **Build** on the left pane of the **Properties** screen, and then set the **Output path** to the same location you used for the `TypeEquivalenceInterface` project, for example, `C:\TypeEquivalenceSample`.
7. Select **Build > Strong naming** on the left pane of the **Properties** screen, and then select the **Sign the assembly** check box. In the **Strong name key file**, select **Browse**.

8. Navigate to and select the *key.snk* file you created in the *TypeEquivalenceInterface* project, and then select **OK**. For more information, see [Create a public-private key pair](#).
9. In **Solution Explorer**, right-click the *TypeEquivalenceRuntime* project and select **Add > Reference**.
10. In the **Reference Manager** dialog, select **Browse** and browse to the output path folder. Select the *TypeEquivalenceInterface.dll* file, select **Add**, and then select **OK**.
11. In **Solution Explorer**, expand the **References** folder and select the *TypeEquivalenceInterface* reference. In the **Properties** pane, set **Specific Version** to **False** if it is not already.
12. Open the *SampleClass* class file in the code editor, and replace its contents with the following code to create the *SampleClass* class:

```
C#  
  
using System;  
using TypeEquivalenceInterface;  
  
namespace TypeEquivalenceRuntime  
{  
    public class SampleClass : ISampleInterface  
    {  
        private string p_UserInput;  
        public string UserInput { get { return p_UserInput; } }  
  
        public void GetUserInput()  
        {  
            Console.WriteLine("Please enter a value:");  
            p_UserInput = Console.ReadLine();  
        }  
    }  
}
```

13. Select **File > Save All** or press **Ctrl + Shift + S** to save the files and project.
14. In **Solution Explorer**, right-click the *TypeEquivalenceRuntime* project and select **Build**. The class library DLL file is compiled and saved to the specified build output path.

Create a client project

Finally, create a type equivalence client program that references the interface assembly.

1. In Visual Studio, select **File > New > Project**.
2. In the **Create a new project** dialog box, type *console* in the **Search for templates** box. Select either the C# or Visual Basic **Console App (.NET Framework)** template from the list, and then select **Next**.
3. In the **Configure your new project** dialog box, under **Project name**, type *TypeEquivalenceClient*, and then select **Create**. The new project is created.
4. In **Solution Explorer**, right-click the **TypeEquivalenceClient** project and select **Properties**.
5. Select **Build** on the left pane of the **Properties** screen, and then set the **Output path** to the same location you used for the **TypeEquivalenceInterface** project, for example, *C:\TypeEquivalenceSample*.
6. In **Solution Explorer**, right-click the **TypeEquivalenceClient** project and select **Add > Reference**.
7. In the **Reference Manager** dialog, if the **TypeEquivalenceInterface.dll** file is already listed, select it. If not, select **Browse**, browse to the output path folder, select the *TypeEquivalenceInterface.dll* file (not the *TypeEquivalenceRuntime.dll*), and select **Add**. Select **OK**.
8. In **Solution Explorer**, expand the **References** folder and select the **TypeEquivalenceInterface** reference. In the **Properties** pane, set **Embed Interop Types** to **True**.
9. Open the *Program.cs* or *Module1.vb* file in the code editor, and replace its contents with the following code to create the client program:

```
C#  
  
using System;  
using System.Reflection;  
using TypeEquivalenceInterface;  
  
namespace TypeEquivalenceClient  
{  
    class Program  
    {  
        static void Main(string[] args)  
        {  
            Assembly sampleAssembly =  
Assembly.Load("TypeEquivalenceRuntime");  
            ISampleInterface sampleClass =  
(ISampleInterface)sampleAssembly.CreateInstance("TypeEquivalenceRuntime")
```

```
    .SampleClass");
        sampleClass.GetUserInput();
        Console.WriteLine(sampleClass.UserInput);

        Console.WriteLine(sampleAssembly.GetName().Version.ToString());
        Console.ReadLine();
    }
}
```

10. Select **File > Save All** or press **Ctrl + Shift + S** to save the files and project.

11. Press **Ctrl + F5** to build and run the program. Note that the console output returns the assembly version 1.0.0.0.

Modify the interface

Now, modify the interface assembly, and change its version.

1. In Visual Studio, select **File > Open > Project/Solution**, and open the **TypeEquivalenceInterface** project.
2. In **Solution Explorer**, right-click the **TypeEquivalenceInterface** project and select **Properties**.
3. Select **Application** on the left pane of the **Properties** screen, and then select **Assembly Information**.
4. In the **Assembly Information** dialog box, change the **Assembly version** and **File version** values to 2.0.0.0, and then select **OK**.
5. Open the *SampleInterface.cs* or *SampleInterface.vb* file, and add the following line of code to the **ISampleInterface** interface:

```
C#
DateTime GetDate();
```

6. Select **File > Save All** or press **Ctrl + Shift + S** to save the files and project.

7. In **Solution Explorer**, right-click the **TypeEquivalenceInterface** project and select **Build**. A new version of the class library DLL file is compiled and saved to the build output path.

Modify the runtime class

Also modify the runtime class and update its version.

1. In Visual Studio, select **File > Open > Project/Solution**, and open the **TypeEquivalenceRuntime** project.
2. In **Solution Explorer**, right-click the **TypeEquivalenceRuntime** project and select **Properties**.
3. Select **Application** on the left pane of the **Properties** screen, and then select **Assembly Information**.
4. In the **Assembly Information** dialog box, change the **Assembly version** and **File version** values to **2.0.0.0**, and then select **OK**.
5. Open the *SampleClass.cs* or *SampleClass.vb* file, and add the following code to the **SampleClass** class:

```
C#  
  
public DateTime GetDate()  
{  
    return DateTime.Now;  
}
```

6. Select **File > Save All** or press **Ctrl + Shift + S** to save the files and project.
7. In **Solution Explorer**, right-click the **TypeEquivalenceRuntime** project and select **Build**. A new version of the class library DLL file is compiled and saved to the build output path.

Run the updated client program

Go to the build output folder location and run *TypeEquivalenceClient.exe*. Note that the console output now reflects the new version of the **TypeEquivalenceRuntime** assembly, **2.0.0.0**, without the program being recompiled.

See also

- [EmbedInteropTypes \(C# Compiler Options\)](#)
- [-link \(Visual Basic\)](#)
- [C# programming guide](#)

- Programming concepts (Visual Basic)
- Assemblies in .NET

 Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

How to: Inspect assembly contents using MetadataLoadContext

Article • 06/30/2022

The reflection API in .NET by default enables developers to inspect the contents of assemblies loaded into the main execution context. However, sometimes it isn't possible to load an assembly into the execution context, for example, because it was compiled for another platform or processor architecture, or it's a [reference assembly](#). The [System.Reflection.MetadataLoadContext](#) API allows you to load and inspect such assemblies. Assemblies loaded into the [MetadataLoadContext](#) are treated only as metadata, that is, you can examine types in the assembly, but you can't execute any code contained in it. Unlike the main execution context, the [MetadataLoadContext](#) doesn't automatically load dependencies from the current directory; instead it uses the custom binding logic provided by the [MetadataAssemblyResolver](#) passed to it.

Prerequisites

To use [MetadataLoadContext](#), install the [System.Reflection.MetadataLoadContext](#) NuGet package. It is supported on any .NET Standard 2.0-compliant target framework, for example, .NET Core 2.0 or .NET Framework 4.6.1.

Create MetadataAssemblyResolver for MetadataLoadContext

Creating the [MetadataLoadContext](#) requires providing the instance of the [MetadataAssemblyResolver](#). The simplest way to provide one is to use the [PathAssemblyResolver](#), which resolves assemblies from the given collection of assembly path strings. This collection, besides assemblies you want to inspect directly, should also include all needed dependencies. For example, to read the custom attribute located in an external assembly, you should include that assembly or an exception will be thrown. In most cases, you should include at least the *core assembly*, that is, the assembly containing built-in system types, such as [System.Object](#). The following code shows how to create the [PathAssemblyResolver](#) using the collection consisting of the inspected assembly and the current runtime's core assembly:

C#

```
var resolver = new PathAssemblyResolver(new string[] {
```

```
"ExampleAssembly.dll", typeof(object).Assembly.Location });
```

If you need access to all BCL types, you can include all runtime assemblies in the collection. The following code shows how to create the [PathAssemblyResolver](#) using the collection consisting of the inspected assembly and all assemblies of the current runtime:

C#

```
// Get the array of runtime assemblies.
string[] runtimeAssemblies =
Directory.GetFiles(RuntimeEnvironment.GetRuntimeDirectory(), "*.dll");

// Create the list of assembly paths consisting of runtime assemblies and
// the inspected assembly.
var paths = new List<string>(runtimeAssemblies);
paths.Add("ExampleAssembly.dll");

// Create PathAssemblyResolver that can resolve assemblies using the created
list.
var resolver = new PathAssemblyResolver(paths);
```

Create MetadataLoadContext

To create the [MetadataLoadContext](#), invoke its constructor

`MetadataLoadContext(MetadataAssemblyResolver, String)`, passing the previously created [MetadataAssemblyResolver](#) as the first parameter and the core assembly name as the second parameter. You can omit the core assembly name, in which case the constructor will attempt to use default names: "mscorlib", "System.Runtime", or "netstandard".

After you've created the context, you can load assemblies into it using methods such as [LoadFromAssemblyPath](#). You can use all reflection APIs on loaded assemblies except ones that involve code execution. The [GetCustomAttributes](#) method does involve the execution of constructors, so use the [GetCustomAttributesData](#) method instead when you need to examine custom attributes in the [MetadataLoadContext](#).

The following code sample creates [MetadataLoadContext](#), loads the assembly into it, and outputs assembly attributes into the console:

C#

```
var mlc = new MetadataLoadContext(resolver);

using (mlc)
```

```

{
    // Load assembly into MetadataLoadContext.
    Assembly assembly = mlc.LoadFromAssemblyPath("ExampleAssembly.dll");
    AssemblyName name = assembly.GetName();

    // Print assembly attribute information.
    Console.WriteLine($"{name.Name} has following attributes: ");

    foreach (CustomAttributeData attr in assembly.GetCustomAttributesData())
    {
        try
        {
            Console.WriteLine(attr.AttributeType);
        }
        catch (FileNotFoundException ex)
        {
            // We are missing the required dependency assembly.
            Console.WriteLine($"Error while getting attribute type:
{ex.Message}");
        }
    }
}

```

If you need to test types in `MetadataLoadContext` for equality or assignability, only use type objects loaded into that context. Mixing `MetadataLoadContext` types with runtime types is not supported. For example, consider a type `testedType` in `MetadataLoadContext`. If you need to test whether another type is assignable from it, don't use code like `typeof(MyType).IsAssignableFrom(testedType)`. Use code like this instead:

C#

```

Assembly matchAssembly =
mlc.LoadFromAssemblyPath(typeof(MyType).Assembly.Location);
Type matchType = assembly.GetType(typeof(MyType).FullName!)!;

if (matchType.IsAssignableFrom(testedType))
{
    Console.WriteLine($"{nameof(matchType)} is assignable from
{nameof(testedType)}");
}

```

Example

For a complete code example, see the [Inspect assembly contents using `MetadataLoadContext` sample](#).

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

Metadata and Self-Describing Components

Article • 03/11/2022

In the past, a software component (.exe or .dll) that was written in one language could not easily use a software component that was written in another language. COM provided a step towards solving this problem. .NET makes component interoperation even easier by allowing compilers to emit additional declarative information into all modules and assemblies. This information, called metadata, helps components to interact seamlessly.

Metadata is binary information describing your program that is stored either in a common language runtime portable executable (PE) file or in memory. When you compile your code into a PE file, metadata is inserted into one portion of the file, and your code is converted to Microsoft intermediate language (MSIL) and inserted into another portion of the file. Every type and member that is defined and referenced in a module or assembly is described within metadata. When code is executed, the runtime loads metadata into memory and references it to discover information about your code's classes, members, inheritance, and so on.

Metadata describes every type and member defined in your code in a language-neutral manner. Metadata stores the following information:

- Description of the assembly.
 - Identity (name, version, culture, public key).
 - The types that are exported.
 - Other assemblies that this assembly depends on.
 - Security permissions needed to run.
- Description of types.
 - Name, visibility, base class, and interfaces implemented.
 - Members (methods, fields, properties, events, nested types).
- Attributes.
 - Additional descriptive elements that modify types and members.

Benefits of Metadata

Metadata is the key to a simpler programming model, and eliminates the need for Interface Definition Language (IDL) files, header files, or any external method of component reference. Metadata enables .NET languages to describe themselves automatically in a language-neutral manner, unseen by both the developer and the user. Additionally, metadata is extensible through the use of attributes. Metadata provides the following major benefits:

- Self-describing files.

Common language runtime modules and assemblies are self-describing. A module's metadata contains everything needed to interact with another module. Metadata automatically provides the functionality of IDL in COM, so you can use one file for both definition and implementation. Runtime modules and assemblies do not even require registration with the operating system. As a result, the descriptions used by the runtime always reflect the actual code in your compiled file, which increases application reliability.

- Language interoperability and easier component-based design.

Metadata provides all the information required about compiled code for you to inherit a class from a PE file written in a different language. You can create an instance of any class written in any managed language (any language that targets the common language runtime) without worrying about explicit marshalling or using custom interoperability code.

- Attributes.

.NET lets you declare specific kinds of metadata, called attributes, in your compiled file. Attributes can be found throughout .NET and are used to control in more detail how your program behaves at run time. Additionally, you can emit your own custom metadata into .NET files through user-defined custom attributes. For more information, see [Attributes](#).

Metadata and the PE File Structure

Metadata is stored in one section of a .NET portable executable (PE) file, while Microsoft intermediate language (MSIL) is stored in another section of the PE file. The metadata portion of the file contains a series of table and heap data structures. The MSIL portion contains MSIL and metadata tokens that reference the metadata portion of the PE file.

You might encounter metadata tokens when you use tools such as the [MSIL Disassembler \(Ildasm.exe\)](#) to view your code's MSIL, for example.

Metadata Tables and Heaps

Each metadata table holds information about the elements of your program. For example, one metadata table describes the classes in your code, another table describes the fields, and so on. If you have ten classes in your code, the class table will have tens rows, one for each class. Metadata tables reference other tables and heaps. For example, the metadata table for classes references the table for methods.

Metadata also stores information in four heap structures: string, blob, user string, and GUID. All the strings used to name types and members are stored in the string heap. For example, a method table does not directly store the name of a particular method, but points to the method's name stored in the string heap.

Metadata Tokens

Each row of each metadata table is uniquely identified in the MSIL portion of the PE file by a metadata token. Metadata tokens are conceptually similar to pointers, persisted in MSIL, that reference a particular metadata table.

A metadata token is a four-byte number. The top byte denotes the metadata table to which a particular token refers (method, type, and so on). The remaining three bytes specify the row in the metadata table that corresponds to the programming element being described. If you define a method in C# and compile it into a PE file, the following metadata token might exist in the MSIL portion of the PE file:

`0x06000004`

The top byte (`0x06`) indicates that this is a **MethodDef** token. The lower three bytes (`000004`) tells the common language runtime to look in the fourth row of the **MethodDef** table for the information that describes this method definition.

Metadata within a PE File

When a program is compiled for the common language runtime, it is converted to a PE file that consists of three parts. The following table describes the contents of each part.

[] Expand table

| PE section | Contents of PE section |
|-------------------|---|
| PE header | The index of the PE file's main sections and the address of the entry point. The runtime uses this information to identify the file as a PE file and to determine where execution starts when loading the program into memory. |
| MSIL instructions | The Microsoft intermediate language instructions (MSIL) that make up your code. Many MSIL instructions are accompanied by metadata tokens. |
| Metadata | Metadata tables and heaps. The runtime uses this section to record information about every type and member in your code. This section also includes custom attributes and security information. |

Run-Time Use of Metadata

To better understand metadata and its role in the common language runtime, it might be helpful to construct a simple program and illustrate how metadata affects its run-time life. The following code example shows two methods inside a class called `MyApp`. The `Main` method is the program entry point, while the `Add` method simply returns the sum of two integer arguments.

```
C#
using System;
public class MyApp
{
    public static int Main()
    {
        int ValueOne = 10;
        int ValueTwo = 20;
        Console.WriteLine("The Value is: {0}", Add(ValueOne, ValueTwo));
        return 0;
    }
    public static int Add(int One, int Two)
    {
        return (One + Two);
    }
}
```

When the code runs, the runtime loads the module into memory and consults the metadata for this class. Once loaded, the runtime performs extensive analysis of the method's Microsoft intermediate language (MSIL) stream to convert it to fast native machine instructions. The runtime uses a just-in-time (JIT) compiler to convert the MSIL instructions to native machine code one method at a time as needed.

The following example shows part of the MSIL produced from the previous code's `Main` function. You can view the MSIL and metadata from any .NET application using the [MSIL Disassembler \(Ildasm.exe\)](#).

```
Console

.entrypoint
.maxstack 3
.locals ([0] int32 ValueOne,
          [1] int32 ValueTwo,
          [2] int32 V_2,
          [3] int32 V_3)
IL_0000: ldc.i4.s 10
IL_0002: stloc.0
IL_0003: ldc.i4.s 20
IL_0005: stloc.1
IL_0006: ldstr      "The Value is: {0}"
IL_000b: ldloc.0
IL_000c: ldloc.1
IL_000d: call int32 ConsoleApplication.MyApp::Add(int32,int32) /* 06000003
*/
```

The JIT compiler reads the MSIL for the whole method, analyzes it thoroughly, and generates efficient native instructions for the method. At `IL_000d`, a metadata token for the `Add` method (`/* 06000003 */`) is encountered and the runtime uses the token to consult the third row of the **MethodDef** table.

The following table shows part of the **MethodDef** table referenced by the metadata token that describes the `Add` method. While other metadata tables exist in this assembly and have their own unique values, only this table is discussed.

| Row | Relative Virtual Address (RVA) | ImplFlags | Flags | Name (Points to string heap.) | Signature (Points to blob heap.) |
|-----|--------------------------------|-----------|-----------|----------------------------------|----------------------------------|
| 1 | 0x00002050 | IL | Public | .ctor (constructor) | |
| | | Managed | ReuseSlot | | |
| | | | | SpecialName | |
| | | | | RTSpecialName | |
| | | | | .ctor | |

| Row | Relative Virtual Address (RVA) | ImplFlags | Flags | Name <small>(Points to string heap.)</small> | Signature (Points to blob heap.) |
|-----|--------------------------------|-----------|---------|---|----------------------------------|
| 2 | 0x00002058 | IL | Public | Main | String |
| | | | Managed | Static | |
| | | | | ReuseSlot | |
| 3 | 0x0000208c | IL | Public | Add | int, int, int |
| | | | Managed | Static | |
| | | | | ReuseSlot | |

Each column of the table contains important information about your code. The **RVA** column allows the runtime to calculate the starting memory address of the MSIL that defines this method. The **ImplFlags** and **Flags** columns contain bitmasks that describe the method (for example, whether the method is public or private). The **Name** column indexes the name of the method from the string heap. The **Signature** column indexes the definition of the method's signature in the blob heap.

The runtime calculates the desired offset address from the **RVA** column in the third row and returns this address to the JIT compiler, which then proceeds to the new address. The JIT compiler continues to process MSIL at the new address until it encounters another metadata token and the process is repeated.

Using metadata, the runtime has access to all the information it needs to load your code and process it into native machine instructions. In this manner, metadata enables self-describing files and, together with the common type system, cross-language inheritance.

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



.NET feedback

.NET is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

System.Reflection.Emit.AssemblyBuilder class

Article • 01/09/2024

This article provides supplementary remarks to the reference documentation for this API.

A dynamic assembly is an assembly that is created using the Reflection Emit APIs. You can use [AssemblyBuilder](#) to generate dynamic assemblies in memory and execute their code during the same application run. A dynamic assembly can reference types defined in another dynamic or static assembly.

To get an [AssemblyBuilder](#) object, use the [AssemblyBuilder.DefineDynamicAssembly](#) method.

Dynamic assemblies can be created using one of the following access modes:

- [AssemblyBuilderAccess.Run](#)

The dynamic assembly represented by an [AssemblyBuilder](#) can be used to execute the emitted code.

- [AssemblyBuilderAccess.RunAndCollect](#)

The dynamic assembly represented by an [AssemblyBuilder](#) can be used to execute the emitted code and is automatically reclaimed by garbage collector.

The access mode must be specified by providing the appropriate [AssemblyBuilderAccess](#) value in the call to the [AssemblyBuilder.DefineDynamicAssembly](#) method when the dynamic assembly is defined and cannot be changed later. The runtime uses the access mode of a dynamic assembly to optimize the assembly's internal representation.

In .NET Framework, you can save dynamic assemblies to files. This feature is not available in .NET Core and .NET 5 and later versions. For an alternative way to generate assembly files, see [MetadataBuilder](#).

In .NET Framework, a dynamic assembly can consist of one or more dynamic modules. If a dynamic assembly contains more than one dynamic module, the assembly's manifest file name should match the module's name that is specified as the first argument to the [DefineDynamicModule](#) method. In .NET Core and .NET 5+, a dynamic assembly can only consist of one dynamic module.

Persistable dynamic assemblies in .NET Framework

In .NET Framework, dynamic assemblies and modules can be saved to files. To support this feature, the [AssemblyBuilderAccess](#) enumeration declares two additional fields: [Save](#) and [RunAndSave](#). The dynamic assembly created using one of these access modes is called a *persistable* assembly, while the regular memory-only assembly is called *transient*.

The dynamic modules in the persistable dynamic assembly are saved when the dynamic assembly is saved using the [Save](#) method. To generate an executable, the [SetEntryPoint](#) method must be called to identify the method that is the entry point to the assembly. Assemblies are saved as DLLs by default, unless the [SetEntryPoint](#) method requests the generation of a console application or a Windows-based application.

Some methods on the base [Assembly](#) class, such as [GetModules](#) and [GetLoadedModules](#), will not work correctly when called from [AssemblyBuilder](#) objects. You can load the defined dynamic assembly and call the methods on the loaded assembly. For example, to ensure that resource modules are included in the returned module list, call [GetModules](#) on the loaded [Assembly](#) object.

The signing of a dynamic assembly using [KeyPair](#) is not effective until the assembly is saved to disk. So, strong names will not work with transient dynamic assemblies.

Dynamic assemblies can reference types defined in another assembly. A transient dynamic assembly can safely reference types defined in another transient dynamic assembly, a persistable dynamic assembly, or a static assembly. However, the common language runtime does not allow a persistable dynamic module to reference a type defined in a transient dynamic module. This is because when the persisted dynamic module is loaded after being saved to disk, the runtime cannot resolve the references to types defined in the transient dynamic module.

Restrictions on emitting to remote application domains in .NET Framework

Some scenarios require a dynamic assembly to be created and executed in a remote application domain. Reflection emit does not allow a dynamic assembly to be emitted directly to a remote application domain. The solution is to emit the dynamic assembly in the current application domain, save the emitted dynamic assembly to disk, and then load the dynamic assembly into the remote application domain. The remoting and application domains are supported only in .NET Framework.

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

.NET is an open source project.
Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

System.Reflection.Emit.MethodBuilder class

Article • 01/09/2024

This article provides supplementary remarks to the reference documentation for this API.

The [MethodBuilder](#) class is used to fully describe a method in Microsoft intermediate language (MSIL), including the name, attributes, signature, and method body. It is used in conjunction with the [TypeBuilder](#) class to create classes at runtime.

You can use reflection emit to define global methods and to define methods as type members. The APIs that define methods return [MethodBuilder](#) objects.

Global methods

A global method is defined by using the [ModuleBuilder.DefineGlobalMethod](#) method, which returns a [MethodBuilder](#) object.

Global methods must be static. If a dynamic module contains global methods, the [ModuleBuilder.CreateGlobalFunctions](#) method must be called before persisting the dynamic module or the containing dynamic assembly because the common language runtime postpones fixing up the dynamic module until all global functions have been defined.

A global native method is defined by using the [ModuleBuilder.DefinePInvokeMethod](#) method. Platform invoke (PInvoke) methods must not be declared abstract or virtual. The runtime sets the [MethodAttributes.PinvokelImpl](#) attribute for a platform invoke method.

Methods as members of types

A method is defined as a type member by using the [TypeBuilder.DefineMethod](#) method, which returns a [MethodBuilder](#) object.

The [DefineParameter](#) method is used to set the name and parameter attributes of a parameter, or of the return value. The [ParameterBuilder](#) object returned by this method represents a parameter or the return value. The [ParameterBuilder](#) object can be used to set the marshaling, to set the constant value, and to apply custom attributes.

Attributes

Members of the [MethodAttributes](#) enumeration define the precise character of a dynamic method:

- Static methods are specified using the [MethodAttributes.Static](#) attribute.
- Final methods (methods that cannot be overridden) are specified using the [MethodAttributes.Final](#) attribute.
- Virtual methods are specified using the [MethodAttributes.Virtual](#) attribute.
- Abstract methods are specified using the [MethodAttributes.Abstract](#) attribute.
- Several attributes determine method visibility. See the description of the [MethodAttributes](#) enumeration.
- Methods that implement overloaded operators must set the [MethodAttributes.SpecialName](#) attribute.
- Finalizers must set the [MethodAttributes.SpecialName](#) attribute.

Known issues

- Although [MethodBuilder](#) is derived from [MethodInfo](#), some of the abstract methods defined in the [MethodInfo](#) class are not fully implemented in [MethodBuilder](#). These [MethodBuilder](#) methods throw the [NotSupportedException](#). For example the [MethodBuilder.Invoke](#) method is not fully implemented. You can reflect on these methods by retrieving the enclosing type using the [Type.GetType](#) or [Assembly.GetType](#) methods.
- Custom modifiers are supported Starting with .NET Framework version 2.0. They are not supported in earlier versions.

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



.NET feedback

.NET is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

System.Reflection.Emit.TypeBuilder class

Article • 01/09/2024

This article provides supplementary remarks to the reference documentation for this API.

[TypeBuilder](#) is the root class used to control the creation of dynamic classes in the runtime. It provides a set of routines that are used to define classes, add methods and fields, and create the class inside a module. A new [TypeBuilder](#) can be created from a dynamic module by calling the [ModuleBuilder.DefineType](#) method, which returns a [TypeBuilder](#) object.

Reflection emit provides the following options for defining types:

- Define a class or interface with the given name.
- Define a class or interface with the given name and attributes.
- Define a class with the given name, attributes, and base class.
- Define a class with the given name, attributes, base class, and the set of interfaces that the class implements.
- Define a class with the given name, attributes, base class, and packing size.
- Define a class with the given name, attributes, base class, and the class size as a whole.
- Define a class with the given name, attributes, base class, packing size, and the class size as a whole.

To create an array type, pointer type, or byref type for an incomplete type that is represented by a [TypeBuilder](#) object, use the [MakeArrayType](#) method, [MakePointerType](#) method, or [MakeByRefType](#) method, respectively.

Before a type is used, the [TypeBuilder.CreateType](#) method must be called. [CreateType](#) completes the creation of the type. Following the call to [CreateType](#), the caller can instantiate the type by using the [Activator.CreateInstance](#) method, and invoke members of the type by using the [Type.InvokeMember](#) method. It is an error to invoke methods that change the implementation of a type after [CreateType](#) has been called. For example, the common language runtime throws an exception if the caller tries to add new members to a type.

A class initializer is created by using the [TypeBuilder.DefineTypeInitializer](#) method. [DefineTypeInitializer](#) returns a [ConstructorBuilder](#) object.

Nested types are defined by calling one of the [TypeBuilder.DefineNestedType](#) methods.

Attributes

The [TypeBuilder](#) class uses the [TypeAttributes](#) enumeration to further specify the characteristics of the type to be created:

- Interfaces are specified using the [TypeAttributes.Interface](#) and [TypeAttributes.Abstract](#) attributes.
- Concrete classes (classes that cannot be extended) are specified using the [TypeAttributes.Sealed](#) attribute.
- Several attributes determine type visibility. See the description of the [TypeAttributes](#) enumeration.
- If [TypeAttributes.SequentialLayout](#) is specified, the class loader lays out fields in the order they are read from metadata. The class loader considers the specified packing size but ignores any specified field offsets. The metadata preserves the order in which the field definitions are emitted. Even across a merge, the metadata will not reorder the field definitions. The loader will honor the specified field offsets only if [TypeAttributes.ExplicitLayout](#) is specified.

Known issues

- Reflection emit does not verify whether a non-abstract class that implements an interface has implemented all the methods declared in the interface. However, if the class does not implement all the methods declared in an interface, the runtime does not load the class.
- Although [TypeBuilder](#) is derived from [Type](#), some of the abstract methods defined in the [Type](#) class are not fully implemented in the [TypeBuilder](#) class. Calls to these [TypeBuilder](#) methods throw a [NotSupportedException](#) exception. The desired functionality can be obtained by retrieving the created type using the [Type.GetType](#) or [Assembly.GetType](#) and reflecting on the retrieved type.

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

 .NET

.NET feedback

.NET is an open source project.
Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

System.Reflection.Emit.DynamicMethod class

Article • 01/09/2024

This article provides supplementary remarks to the reference documentation for this API.

You can use the [DynamicMethod](#) class to generate and execute a method at run time, without having to generate a dynamic assembly and a dynamic type to contain the method. The executable code created by the just-in-time (JIT) compiler is reclaimed when the [DynamicMethod](#) object is reclaimed. Dynamic methods are the most efficient way to generate and execute small amounts of code.

A dynamic method can be anonymously hosted, or it can be logically associated with a module or with a type.

- If the dynamic method is anonymously hosted, it is located in a system-provided assembly, and therefore is isolated from other code. By default, it does not have access to any non-public data. An anonymously hosted dynamic method can have restricted ability to skip the JIT compiler's visibility checks, if it has been granted [ReflectionPermission](#) with the [ReflectionPermissionFlag.RestrictedMemberAccess](#) flag. The trust level of the assembly whose non-public members are accessed by the dynamic method must be equal to, or a subset of, the trust level of the call stack that emitted the dynamic method. For more information about anonymously hosted dynamic methods, see [Walkthrough: Emitting Code in Partial Trust Scenarios](#).
- If the dynamic method is associated with a module that you specify, the dynamic method is effectively global to that module. It can access all types in the module and all `internal` (`Friend` in Visual Basic) members of the types. You can associate a dynamic method with any module, regardless of whether you created the module, provided that a demand for [ReflectionPermission](#) with the [RestrictedMemberAccess](#) flag can be satisfied by the call stack that includes your code. If the [ReflectionPermissionFlag.MemberAccess](#) flag is included in the grant, the dynamic method can skip the JIT compiler's visibility checks and access the private data of all types declared in the module or in any other module in any assembly.

 **Note**

When you specify the module with which a dynamic method is associated, that module must not be in the system-provided assembly that is used for anonymous hosting.

- If the dynamic method is associated with a type that you specify, it has access to all members of the type, regardless of access level. In addition, JIT visibility checks can be skipped. This gives the dynamic method access to the private data of other types declared in the same module or in any other module in any assembly. You can associate a dynamic method with any type, but your code must be granted [ReflectionPermission](#) with both the [RestrictedMemberAccess](#) and [MemberAccess](#) flags.

The following table shows which types and members are accessible to an anonymously hosted dynamic method, with and without JIT visibility checks, depending on whether [ReflectionPermission](#) with the [RestrictedMemberAccess](#) flag is granted.

[+] Expand table

| Visibility checks | Without | With RestrictedMemberAccess |
|---|---|--|
| | RestrictedMemberAccess | |
| Without skipping JIT visibility checks | Public members of public types in any assembly. | Public members of public types in any assembly. |
| Skipping JIT visibility checks, with restrictions | Public members of public types in any assembly. | All members of all types, only in assemblies whose trust levels are equal to or less than the trust level of the assembly that emitted the dynamic method. |

The following table shows which types and members are accessible to a dynamic method that's associated with a module or with a type in a module.

[+] Expand table

| Skip JIT visibility checks | Associated with module | Associated with type |
|----------------------------|---|---|
| No | Public and internal members of public, internal, and private types in the module. | All members of the associated type. Public and internal members of all the other types in the module. |
| | Public members of public types in any assembly. | Public members of public types in any assembly. |

| Skip JIT visibility checks | Associated with module | Associated with type |
|----------------------------|---|---|
| Yes | All members of all types in any assembly. | All members of all types in any assembly. |

A dynamic method that is associated with a module has the permissions of that module. A dynamic method that is associated with a type has the permissions of the module containing that type.

Dynamic methods and their parameters do not have to be named, but you can specify names to assist in debugging. Custom attributes are not supported on dynamic methods or their parameters.

Although dynamic methods are `static` methods (`Shared` methods in Visual Basic), the relaxed rules for delegate binding allow a dynamic method to be bound to an object, so that it acts like an instance method when called using that delegate instance. An example that demonstrates this is provided for the [CreateDelegate\(Type, Object\)](#) method overload.

Verification

The following list summarizes the conditions under which dynamic methods can contain unverifiable code. (For example, a dynamic method is unverifiable if its [InitLocals](#) property is set to `false`.)

- A dynamic method that's associated with a security-critical assembly is also security-critical, and can skip verification. For example, an assembly without security attributes that is run as a desktop application is treated as security-critical by the runtime. If you associate a dynamic method with the assembly, the dynamic method can contain unverifiable code.
- If a dynamic method that contains unverifiable code is associated with an assembly that has level 1 transparency, the just-in-time (JIT) compiler injects a security demand. The demand succeeds only if the dynamic method is executed by fully trusted code. See [Security-Transparent Code, Level 1](#).
- If a dynamic method that contains unverifiable code is associated with an assembly that has level 2 transparency (such as mscorlib.dll), it throws an exception (injected by the JIT compiler) instead of making a security demand. See [Security-Transparent Code, Level 2](#).
- An anonymously hosted dynamic method that contains unverifiable code always throws an exception. It can never skip verification, even if it is created and

executed by fully trusted code.

The exception that's thrown for unverifiable code varies depending on the way the dynamic method is invoked. If you invoke a dynamic method by using a delegate returned from the [CreateDelegate](#) method, a [VerificationException](#) is thrown. If you invoke the dynamic method by using the [Invoke](#) method, a [TargetInvocationException](#) is thrown with an inner [VerificationException](#).

 [Collaborate with us on GitHub](#)

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

 .NET

.NET feedback

.NET is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

System.Reflection.PortableExecutable.DebugDirectoryEntryType enum

Article • 01/09/2024

This article provides supplementary remarks to the reference documentation for this API.

The [DebugDirectoryEntryType](#) enum describes the format of the debugging information of a [DebugDirectoryEntry](#).

See the following for the specifications related to individual enumeration members:

[] Expand table

| Member | Specification |
|---------------------|---|
| CodeView | CodeView Debug Directory Entry (type 2) ↗ |
| EmbeddedPortablePdb | Embedded Portable PDB Debug Directory Entry (type 17) ↗ |
| PdbChecksum | PDB Checksum Debug Directory Entry (type 19) ↗ |
| Reproducible | See Deterministic Debug Directory Entry (type 16) ↗ |

DebugDirectoryEntryType.Reproducible

The tool that produced the deterministic PE/COFF file guarantees that the entire content of the file is based solely on documented inputs given to the tool (such as source files, resource files, compiler options, etc.) rather than ambient environment variables (such as the current time, the operating system, the bitness of the process running the tool, etc.).

The value of the `TimeDateStamp` field in the COFF File Header of a deterministic PE/COFF file does not indicate the date and time when the file was produced and should not be interpreted that way. Instead, the value of the field is derived from a hash of the file content. The algorithm to calculate this value is an implementation detail of the tool that produced the file.

The debug directory entry of type [Reproducible](#) must have all fields except for [DebugDirectoryEntry.Type](#) zeroed.

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

.NET is an open source project.
Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

Dependency loading in .NET

Article • 08/30/2022

Every .NET application has dependencies. Even the simple `hello world` app has dependencies on portions of the .NET class libraries.

Understanding the default assembly loading logic in .NET can help you troubleshoot typical deployment issues.

In some applications, dependencies are dynamically determined at run time. In these situations, it's critical to understand how managed assemblies and unmanaged dependencies are loaded.

AssemblyLoadContext

The [AssemblyLoadContext](#) API is central to the .NET loading design. The [Understanding AssemblyLoadContext](#) article provides a conceptual overview of the design.

Loading details

The loading algorithm details are covered briefly in several articles:

- [Managed assembly loading algorithm](#)
- [Satellite assembly loading algorithm](#)
- [Unmanaged \(native\) library loading algorithm](#)
- [Default probing](#)

Create an app with plugins

The tutorial [Create a .NET application with plugins](#) describes how to create a custom `AssemblyLoadContext`. It uses an `AssemblyDependencyResolver` to resolve the dependencies of the plugin. The tutorial correctly isolates the plugin's dependencies from the hosting application.

Assembly unloadability

The [How to use and debug assembly unloadability in .NET](#) article is a step-by-step tutorial. It shows how to load a .NET application, execute it, and then unload it. The article also provides debugging tips.

Collect detailed assembly loading information

The [Collect detailed assembly loading information](#) article describes how to collect detailed information about managed assembly loading in the runtime. It uses the `dotnet-trace` tool to capture assembly loader events in a trace of a running process.

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

About System.Runtime.Loader.AssemblyLoadContext

Article • 08/30/2022

The [AssemblyLoadContext](#) class was introduced in .NET Core and is not available in .NET Framework. This article supplements the [AssemblyLoadContext](#) API documentation with conceptual information.

This article is relevant to developers implementing dynamic loading, especially dynamic-loading framework developers.

What is the AssemblyLoadContext?

Every .NET 5+ and .NET Core application implicitly uses [AssemblyLoadContext](#). It's the runtime's provider for locating and loading dependencies. Whenever a dependency is loaded, an [AssemblyLoadContext](#) instance is invoked to locate it.

- AssemblyLoadContext provides a service of locating, loading, and caching managed assemblies and other dependencies.
- To support dynamic code loading and unloading, it creates an isolated context for loading code and its dependencies in their own [AssemblyLoadContext](#) instance.

Versioning rules

A single [AssemblyLoadContext](#) instance is limited to loading exactly one version of an [Assembly](#) per [simple assembly name](#). When an assembly reference is resolved against an [AssemblyLoadContext](#) instance that already has an assembly of that name loaded, the requested version is compared to the loaded version. The resolution will succeed only if the loaded version is equal or higher to the requested version.

When do you need multiple AssemblyLoadContext instances?

The restriction that a single [AssemblyLoadContext](#) instance can load only one version of an assembly can become a problem when loading code modules dynamically. Each module is independently compiled, and the modules may depend on different versions

of an [Assembly](#). This is often a problem when different modules depend on different versions of a commonly used library.

To support dynamically loading code, the [AssemblyLoadContext](#) API provides for loading conflicting versions of an [Assembly](#) in the same application. Each [AssemblyLoadContext](#) instance provides a unique dictionary that maps each [AssemblyName.Name](#) to a specific [Assembly](#) instance.

It also provides a convenient mechanism for grouping dependencies related to a code module for later unload.

The [AssemblyLoadContext.Default](#) instance

The [AssemblyLoadContext.Default](#) instance is automatically populated by the runtime at startup. It uses [default probing](#) to locate and find all static dependencies.

It solves the most common dependency loading scenarios.

Dynamic dependencies

[AssemblyLoadContext](#) has various events and virtual functions that can be overridden.

The [AssemblyLoadContext.Default](#) instance only supports overriding the events.

The articles [Managed assembly loading algorithm](#), [Satellite assembly loading algorithm](#), and [Unmanaged \(native\) library loading algorithm](#) refer to all the available events and virtual functions. The articles show each event and function's relative position in the loading algorithms. This article doesn't reproduce that information.

This section covers the general principles for the relevant events and functions.

- **Be repeatable.** A query for a specific dependency must always result in the same response. The same loaded dependency instance must be returned. This requirement is fundamental for cache consistency. For managed assemblies in particular, we're creating an [Assembly](#) cache. The cache key is a simple assembly name, [AssemblyName.Name](#).
- **Typically don't throw.** It's expected that these functions return `null` rather than throw when unable to find the requested dependency. Throwing will prematurely end the search and propagate an exception to the caller. Throwing should be restricted to unexpected errors like a corrupted assembly or an out of memory condition.

- **Avoid recursion.** Be aware that these functions and handlers implement the loading rules for locating dependencies. Your implementation shouldn't call APIs that trigger recursion. Your code should typically call `AssemblyLoadContext` load functions that require a specific path or memory reference argument.
- **Load into the correct `AssemblyLoadContext`.** The choice of where to load dependencies is application-specific. The choice is implemented by these events and functions. When your code calls `AssemblyLoadContext` load-by-path functions call them on the instance where you want the code loaded. Sometime returning `null` and letting the `AssemblyLoadContext.Default` handle the load may be the simplest option.
- **Be aware of thread races.** Loading can be triggered by multiple threads. The `AssemblyLoadContext` handles thread races by atomically adding assemblies to its cache. The race loser's instance is discarded. In your implementation logic, don't add extra logic that doesn't handle multiple threads properly.

How are dynamic dependencies isolated?

Each `AssemblyLoadContext` instance represents a unique scope for `Assembly` instances and `Type` definitions.

There's no binary isolation between these dependencies. They're only isolated by not finding each other by name.

In each `AssemblyLoadContext`:

- `AssemblyName.Name` may refer to a different `Assembly` instance.
- `Type.GetType` may return a different type instance for the same type `name`.

Shared dependencies

Dependencies can easily be shared between `AssemblyLoadContext` instances. The general model is for one `AssemblyLoadContext` to load a dependency. The other shares the dependency by using a reference to the loaded assembly.

This sharing is required of the runtime assemblies. These assemblies can only be loaded into the `AssemblyLoadContext.Default`. The same is required for frameworks like `ASP.NET`, `WPF`, or `WinForms`.

It's recommended that shared dependencies be loaded into `AssemblyLoadContext.Default`. This sharing is the common design pattern.

Sharing is implemented in the coding of the custom [AssemblyLoadContext](#) instance. [AssemblyLoadContext](#) has various events and virtual functions that can be overridden. When any of these functions return a reference to an [Assembly](#) instance that was loaded in another [AssemblyLoadContext](#) instance, the [Assembly](#) instance is shared. The standard load algorithm defers to [AssemblyLoadContext.Default](#) for loading to simplify the common sharing pattern. For more information, see [Managed assembly loading algorithm](#).

Type-conversion issues

When two [AssemblyLoadContext](#) instances contain type definitions with the same `name`, they're not the same type. They're the same type if and only if they come from the same [Assembly](#) instance.

To complicate matters, exception messages about these mismatched types can be confusing. The types are referred to in the exception messages by their simple type names. The common exception message in this case is of the form:

Object of type 'IsolatedType' cannot be converted to type 'IsolatedType'.

Debug type-conversion issues

Given a pair of mismatched types, it's important to also know:

- Each type's [Type.Assembly](#).
- Each type's [AssemblyLoadContext](#), which can be obtained via the [AssemblyLoadContext.GetLoadContext\(Assembly\)](#) function.

Given two objects `a` and `b`, evaluating the following in the debugger will be helpful:

C#

```
// In debugger look at each assembly's instance, Location, and FullName
a.GetType().Assembly
b.GetType().Assembly
// In debugger look at each AssemblyLoadContext's instance and name
System.Runtime.Loader.AssemblyLoadContext.GetLoadContext(a.GetType().Assembly)
System.Runtime.Loader.AssemblyLoadContext.GetLoadContext(b.GetType().Assembly)
```

Resolve type-conversion issues

There are two design patterns for solving these type conversion issues.

1. Use common shared types. This shared type can either be a primitive runtime type, or it can involve creating a new shared type in a shared assembly. Often the shared type is an [interface](#) defined in an application assembly. For more information, read about [how dependencies are shared](#).
2. Use marshalling techniques to convert from one type to another.

 **Collaborate with us on GitHub**

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

 .NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

Default probing

Article • 10/11/2022

The [AssemblyLoadContext.Default](#) instance is responsible for locating an assembly's dependencies. This article describes the [AssemblyLoadContext.Default](#) instance's probing logic.

Host configured probing properties

When the runtime is started, the runtime host provides a set of named probing properties that configure [AssemblyLoadContext.Default](#) probe paths.

Each probing property is optional. If present, each property is a string value that contains a delimited list of absolute paths. The delimiter is ';' on Windows and ':' on all other platforms.

| Property Name | Description |
|-------------------------------|--|
| TRUSTED_PLATFORM_ASSEMBLIES | List of platform and application assembly file paths. |
| PLATFORM_RESOURCE_ROOTS | List of directory paths to search for satellite resource assemblies. |
| NATIVE_DLL_SEARCH_DIRECTORIES | List of directory paths to search for unmanaged (native) libraries. |
| APP_PATHS | List of directory paths to search for managed assemblies. |

How are the properties populated?

There are two main scenarios for populating the properties depending on whether the `<myapp>.deps.json` file exists.

- When the `*.deps.json` file is present, it's parsed to populate the probing properties.
- When the `*.deps.json` file isn't present, the application's directory is assumed to contain all the dependencies. The directory's contents are used to populate the probing properties.

Additionally, the `*.deps.json` files for any referenced frameworks are similarly parsed.

The environment variable `DOTNET_ADDITIONAL_DEPS` can be used to add additional dependencies. `dotnet.exe` also contains an optional `--additional-deps` parameter to set this value on application startup.

The `APP_PATHS` property is not populated by default and is omitted for most applications.

The list of all `*.deps.json` files used by the application can be accessed via
`System.AppContext.GetData("APP_CONTEXT_DEPS_FILES")`.

How do I see the probing properties from managed code?

Each property is available by calling the `AppContext.GetData(String)` function with the property name from the table above.

How do I debug the probing properties' construction?

The .NET Core runtime host will output useful trace messages when certain environment variables are enabled:

| Environment Variable | Description |
|--|--|
| <code>COREHOST_TRACE=1</code> | Enables tracing. |
| <code>COREHOST_TRACEFILE=<path></code> | Traces to a file path instead of the default <code>stderr</code> . |
| <code>COREHOST_TRACE_VERBOSITY</code> | Sets the verbosity from 1 (lowest) to 4 (highest). |

Managed assembly default probing

When probing to locate a managed assembly, the `AssemblyLoadContext.Default` looks in order at:

- Files matching the `AssemblyName.Name` in `TRUSTED_PLATFORM_ASSEMBLIES` (after removing file extensions).
- Assembly files in `APP_PATHS` with common file extensions.

Satellite (resource) assembly probing

To find a satellite assembly for a specific culture, construct a set of file paths.

For each path in `PLATFORM_RESOURCE_ROOTS` and then `APP_PATHS`, append the `CultureInfo.Name` string, a directory separator, the `AssemblyName.Name` string, and the extension '.dll'.

If any matching file exists, attempt to load and return it.

Unmanaged (native) library probing

The runtime's unmanaged library probing algorithm is identical on all platforms. However, since the actual load of the unmanaged library is performed by the underlying platform, the observed behavior can be slightly different.

1. Check if the supplied library name represents an absolute or relative path.
2. If the name represents an absolute path, use the name directly for all subsequent operations. Otherwise, use the name and create platform-defined combinations to consider. Combinations consist of platform specific prefixes (for example, `lib`) and/or suffixes (for example, `.dll`, `.dylib`, and `.so`). This is not an exhaustive list, and it doesn't represent the exact effort made on each platform. It's just an example of what is considered. For more information, see [native library loading](#).
3. The name and, if the path is relative, each combination, is then used in the following steps. The first successful load attempt immediately returns the handle to the loaded library.
 - Append it to each path supplied in the `NATIVE_DLL_SEARCH_DIRECTORIES` property and attempt to load.
 - If `DefaultDllImportSearchPathsAttribute` is either not defined on the calling assembly or p/invoke or is defined and includes `DllImportSearchPath.AssemblyDirectory`, append the name or combination to the calling assembly's directory and attempt to load.
 - Use it directly to load the library.
4. Indicate that the library failed to load.

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

more information, see [our contributor guide](#).

Managed assembly loading algorithm

Article • 08/30/2022

Managed assemblies are located and loaded with an algorithm that has various stages.

All managed assemblies except satellite assemblies and `WinRT` assemblies use the same algorithm.

When are managed assemblies loaded?

The most common mechanism to trigger a managed assembly load is a static assembly reference. These references are inserted by the compiler whenever code uses a type defined in another assembly. These assemblies are loaded (`load-by-name`) as needed by the runtime. The exact timing of when the static assembly references are loaded is unspecified. It can vary between runtime versions and is influenced by optimizations like inlining.

The direct use of the following APIs will also trigger loads:

| API | Description | Active <code>AssemblyLoadContext</code> |
|--|---|--|
| AssemblyLoadContext.LoadFromAssemblyName | <code>Load-by-name</code> | The <code>this</code> instance. |
| AssemblyLoadContext.LoadFromAssemblyPath AssemblyLoadContext.LoadFromNativeImagePath | Load from path. | The <code>this</code> instance. |
| AssemblyLoadContext.LoadFromStream | Load from object. | The <code>this</code> instance. |
| Assembly.LoadFile | Load from path in a new <code>AssemblyLoadContext</code> instance | The new <code>AssemblyLoadContext</code> instance. |
| Assembly.LoadFrom | Load from path in the <code>AssemblyLoadContext.Default</code> instance. Adds an <code>AppDomain.AssemblyResolve</code> handler. The handler will load the assembly's dependencies from its directory. | The <code>AssemblyLoadContext.Default</code> instance. |
| Assembly.Load(AssemblyName) Assembly.Load(String) Assembly.LoadWithPartialName | <code>Load-by-name</code> . | Inferred from caller. Prefer <code>AssemblyLoadContext</code> methods. |
| Assembly.Load(Byte[]) Assembly.Load(Byte[], Byte[]) | Load from object in a new <code>AssemblyLoadContext</code> instance. | The new <code>AssemblyLoadContext</code> instance. |

| API | Description | Active | AssemblyLoadContext |
|--|---|---|---------------------|
| Type.GetType(String) | Load-by-name. | Inferred from caller. | |
| Type.GetType(String, Boolean) | | Prefer Type.GetType methods with an <code>assemblyResolver</code> argument. | |
| Type.GetType(String, Boolean, Boolean) | | | |
| Assembly.GetType | If type <code>name</code> describes an assembly qualified generic type, trigger a Load-by-name. | Inferred from caller. | |
| | | Prefer Type.GetType when using assembly qualified type names. | |
| Activator.CreateInstance(String, String) | Load-by-name. | Inferred from caller. | |
| Activator.CreateInstance(String, String, Object[]) | | Prefer | |
| Activator.CreateInstance(String, String, Boolean, | | Activator.CreateInstance methods taking a <code>Type</code> argument. | |
| BindingFlags, Binder, Object[], CultureInfo, | | | |
| Object[]) | | | |

Algorithm

The following algorithm describes how the runtime loads a managed assembly.

1. Determine the active [AssemblyLoadContext](#).

- For a static assembly reference, the active [AssemblyLoadContext](#) is the instance that loaded the referring assembly.
- Preferred APIs make the active [AssemblyLoadContext](#) explicit.
- Other APIs infer the active [AssemblyLoadContext](#). For these APIs, the [AssemblyLoadContext.CurrentContextualReflectionContext](#) property is used. If its value is `null`, then the inferred [AssemblyLoadContext](#) instance is used.
- See the table in the [When are managed assemblies loaded?](#) section.

2. For the Load-by-name methods, the active [AssemblyLoadContext](#) loads the assembly in the following priority order:

- Check its cache-by-name.
- Call the [AssemblyLoadContext.Load](#) function.
- Check the [AssemblyLoadContext.Default](#) instance's cache and run managed assembly default probing logic. If an assembly is newly loaded, a reference is added to the [AssemblyLoadContext.Default](#) instance's cache-by-name.
- Raise the [AssemblyLoadContext.Resolving](#) event for the active AssemblyLoadContext.
- Raise the [AppDomain.AssemblyResolve](#) event.

3. For the other types of loads, the active [AssemblyLoadContext](#) loads the assembly in the following priority order:

- Check its cache-by-name.

- Load from the specified path or raw assembly object. If an assembly is newly loaded, a reference is added to the `active AssemblyLoadContext` instance's `cache-by-name`.

4. In either case, if an assembly is newly loaded, then the `AppDomain.AssemblyLoad` event is raised.

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

Satellite assembly loading algorithm

Article • 11/11/2023

Satellite assemblies are used to store localized resources customized for language and culture.

Satellite assemblies use a different loading algorithm than general managed assemblies.

When are satellite assemblies loaded?

Satellite assemblies are loaded when loading a localized resource.

The basic API to load localized resources is the [System.Resources.ResourceManager](#) class. Ultimately the [ResourceManager](#) class will call the [GetSatelliteAssembly](#) method for each [CultureInfo.Name](#).

Higher-level APIs may abstract the low-level API.

Algorithm

The .NET Core resource fallback process involves the following steps:

1. Determine the `active` [AssemblyLoadContext](#) instance. In all cases, the `active` instance is the executing assembly's [AssemblyLoadContext](#).
2. The `active` instance loads a satellite assembly for the requested culture in the following priority order:
 - Check its cache.
 - If `active` is the [AssemblyLoadContext.Default](#) instance, run the [default satellite \(resource\) assembly probing](#) logic.
 - Call the [AssemblyLoadContext.Load](#) function.
 - If the managed assembly corresponding to the satellite assembly was loaded from a file, check the directory of the managed assembly for a subdirectory that matches the requested [CultureInfo.Name](#) (for example, `es-MX`).

Note

On Linux and macOS, the subdirectory is case-sensitive and must either:

- Exactly match case.
 - Be in lower case.
- Raise the [AssemblyLoadContext.Resolving](#) event.
 - Raise the [AppDomain.AssemblyResolve](#) event.

3. If a satellite assembly is loaded:

- The [AppDomain.AssemblyLoad](#) event is raised.
- The assembly is searched for the requested resource. If the runtime finds the resource in the assembly, it uses it. If it doesn't find the resource, it continues the search.

 **Note**

To find a resource within the satellite assembly, the runtime searches for the resource file requested by the [ResourceManager](#) for the current [CultureInfo.Name](#). Within the resource file, it searches for the requested resource name. If either is not found, the resource is treated as not found.

4. The [ResourceManager](#) next searches the parent culture assemblies through many potential levels, each time repeating steps 2 & 3.

Each culture has only one parent, which is defined by the [CultureInfo.Parent](#) property.

The search for parent cultures stops when a culture's [Parent](#) property is [CultureInfo.InvariantCulture](#).

For the [InvariantCulture](#), we don't return to steps 2 & 3, but rather continue with step 5.

5. If the resource is still not found, the [ResourceManager](#) uses the resource for the default (fallback) culture.

Typically, the resources for the default culture are included in the main application assembly. However, you can specify [UltimateResourceFallbackLocation.Satellite](#) for the [NeutralResourcesLanguageAttribute.Location](#) property. This value indicates that the ultimate fallback location for resources is a satellite assembly rather than the main assembly.

Note

The default culture is the ultimate fallback. Therefore, we recommend that you always include an exhaustive set of resources in the default resource file. This helps prevent exceptions from being thrown. By having an exhaustive set, you provide a fallback for all resources and ensure that at least one resource is always present for the user, even if it is not culturally specific.

6. Finally,

- If the runtime doesn't find a resource file for a default (fallback) culture, a [MissingManifestResourceException](#) or [MissingSatelliteAssemblyException](#) exception is thrown.
- If the resource file is found but the requested resource isn't present, the request returns `null`.

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

Unmanaged (native) library loading algorithm

Article • 10/27/2021

Unmanaged libraries are located and loaded with an algorithm involving various stages.

The following algorithm describes how native libraries are loaded through `PInvoke`.

`PInvoke` load library algorithm

`PInvoke` uses the following algorithm when attempting to load an unmanaged assembly:

1. Determine the `active AssemblyLoadContext`. For an unmanaged load library, the `active AssemblyLoadContext` is the one with the assembly that defines the `PInvoke`.
2. For the `active AssemblyLoadContext`, try to find the assembly in priority order by:
 - Checking its cache.
 - Calling the current `System.Runtime.InteropServices.DllImportResolver` delegate set by the `NativeLibrary.SetDllImportResolver(Assembly, DllImportResolver)` function.
 - Calling the `AssemblyLoadContext.LoadUnmanagedDll` function on the `active AssemblyLoadContext`.
 - Checking the `AppDomain` instance's cache and running the `Unmanaged (native) library probing` logic.
 - Raising the `AssemblyLoadContext.ResolvingUnmanagedDll` event for the `active AssemblyLoadContext`.

 Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 Open a documentation issue

issues and pull requests. For more information, see [our contributor guide](#).

 [Provide product feedback](#)

Collect detailed assembly loading information

Article • 11/08/2021

Starting with .NET 5, the runtime can emit events through `EventPipe` with detailed information about [managed assembly loading](#) to aid in diagnosing assembly loading issues. These `events` are emitted by the `Microsoft-Windows-DotNERTRuntime` provider under the `AssemblyLoader` keyword (`0x4`).

Prerequisites

- [.NET 5 SDK](#) or later versions
- `dotnet-trace` tool

ⓘ Note

The scope of `dotnet-trace` capabilities is greater than collecting detailed assembly loading information. For more information on the usage of `dotnet-trace`, see [dotnet-trace](#).

Collect a trace with assembly loading events

You can use `dotnet-trace` to trace an existing process or to launch a child process and trace it from startup.

Trace an existing process

To enable assembly loading events in the runtime and collect a trace of them, use `dotnet-trace` with the following command:

Console

```
dotnet-trace collect --providers Microsoft-Windows-DotNERTRuntime:4 --process-id <pid>
```

This command collects a trace of the specified `<pid>`, enabling the `AssemblyLoader` events in the `Microsoft-Windows-DotNERTRuntime` provider. The result is a `.nettrace` file.

Use dotnet-trace to launch a child process and trace it from startup

Sometimes it may be useful to collect a trace of a process from its startup. For apps running .NET 5 or later, you can use `dotnet-trace` to do this.

The following command launches `hello.exe` with `arg1` and `arg2` as its command line arguments and collects a trace from its runtime startup:

Console

```
dotnet-trace collect --providers Microsoft-Windows-DotNERTRuntime:4 -- hello.exe arg1 arg2
```

You can stop collecting the trace by pressing `Enter` or `Ctrl + C`. This also closes `hello.exe`.

ⓘ Note

- Launching `hello.exe` via `dotnet-trace` redirects its input and output, and you won't be able to interact with it on the console by default. Use the `--show-child-io` switch to interact with its `stdin` and `stdout`.
- Exiting the tool via `Ctrl + C` or `SIGTERM` safely ends both the tool and the child process.
- If the child process exits before the tool, the tool exits as well and the trace should be safely viewable.

View a trace

The collected trace file can be viewed on Windows using the Events view in [PerfView](#). All the assembly loading events will be prefixed with `Microsoft-Windows-DotNERTRuntime/AssemblyLoader`.

Example (on Windows)

This example uses the [assembly loading extension points sample](#). The application attempts to load an assembly `MyLibrary` - an assembly that is not referenced by the application and thus requires handling in an assembly loading extension point to be successfully loaded.

Collect the trace

1. Navigate to the directory with the downloaded sample. Build the application with:

```
Console  
dotnet build
```

2. Launch the application with arguments indicating that it should pause, waiting for a key press. On resuming, it will attempt to load the assembly in the default `AssemblyLoadContext` - without the handling necessary for a successful load. Navigate to the output directory and run:

```
Console  
AssemblyLoading.exe /d default
```

3. Find the application's process ID.

```
Console  
dotnet-trace ps
```

The output will list the available processes. For example:

```
Console  
35832 AssemblyLoading C:\src\AssemblyLoading\bin\Debug\net5.0\AssemblyLoading.exe
```

4. Attach `dotnet-trace` to the running application.

```
Console  
dotnet-trace collect --providers Microsoft-Windows-DotNETRuntime:4 --process-id 35832
```

5. In the window running the application, press any key to let the program continue. Tracing will automatically stop once the application exits.

View the trace

Open the collected trace in [PerfView](#) and open the Events view. Filter the events list to `Microsoft-Windows-DotNETRuntime/AssemblyLoader` events.

| Event Types | Filter: AssemblyLoader |
|---|------------------------|
| Microsoft-Windows-DotNETRuntime/AssemblyLoader/AppDomainAssemblyResolveHandlerInvoked | |
| Microsoft-Windows-DotNETRuntime/AssemblyLoader/AssemblyLoadContextResolvingHandlerInvoked | |
| Microsoft-Windows-DotNETRuntime/AssemblyLoader/KnownPathProbed | |
| Microsoft-Windows-DotNETRuntime/AssemblyLoader/ResolutionAttempted | |
| Microsoft-Windows-DotNETRuntime/AssemblyLoader/Start | |
| Microsoft-Windows-DotNETRuntime/AssemblyLoader/Stop | |

All assembly loads that occurred in the application after tracing started will be shown. To inspect the load operation for the assembly of interest for this example - `MyLibrary`, we can do some more filtering.

Assembly loads

Filter the view to the `Start` and `Stop` events under `Microsoft-Windows-DotNETRuntime/AssemblyLoader` using the event list on the left. Add the columns `AssemblyName`, `ActivityID`, and `Success` to the view. Filter to events containing `MyLibrary`.

| Text Filter: | MyLibrary | Columns To Display: | Cols | AssemblyName ActivityID Success * |
|--|---|---------------------|---------|-----------------------------------|
| Histogram: | | | | |
| Event Name | AssemblyName | ActivityID | Success | |
| Microsoft-Windows-DotNETRuntime/AssemblyLoader/Start | MyLibrary, Culture=neutral, PublicKeyToken=null | //1/2/ | | |
| Microsoft-Windows-DotNETRuntime/AssemblyLoader/Stop | MyLibrary, Culture=neutral, PublicKeyToken=null | //1/2/ | False | |

| Event Name | AssemblyName | ActivityID | Success |
|----------------------|---|------------|---------|
| AssemblyLoader/Start | MyLibrary, Culture=neutral, PublicKeyToken=null | //1/2/ | |
| AssemblyLoader/Stop | MyLibrary, Culture=neutral, PublicKeyToken=null | //1/2/ | False |

You should see one `Start`/`Stop` pair with `Success=False` on the `Stop` event, indicating the load operation failed. Note that the two events have the same activity ID. The activity ID can be used to filter all the other assembly loader events to just the ones corresponding to this load operation.

Breakdown of attempt to load

For a more detailed breakdown of the load operation, filter the view to the `ResolutionAttempted` events under `Microsoft-Windows-DotNETRuntime/AssemblyLoader` using the event list on the left. Add the columns `AssemblyName`, `Stage`, and `Result` to the view. Filter to events with the activity ID from the `Start`/`Stop` pair.

| Text Filter: | //1/2/ | Columns To Display: | Cols | AssemblyName Stage Result * |
|--|---|-----------------------------------|------------------|-----------------------------|
| Histogram: | | | | |
| Event Name | AssemblyName | Stage | Result | |
| Microsoft-Windows-DotNETRuntime/AssemblyLoader/ResolutionAttempted | MyLibrary, Culture=neutral, PublicKeyToken=null | FindInLoadContext | AssemblyNotFound | |
| Microsoft-Windows-DotNETRuntime/AssemblyLoader/ResolutionAttempted | MyLibrary, Culture=neutral, PublicKeyToken=null | ApplicationAssemblies | AssemblyNotFound | |
| Microsoft-Windows-DotNETRuntime/AssemblyLoader/ResolutionAttempted | MyLibrary, Culture=neutral, PublicKeyToken=null | AssemblyLoadContextResolvingEvent | AssemblyNotFound | |
| Microsoft-Windows-DotNETRuntime/AssemblyLoader/ResolutionAttempted | MyLibrary, Culture=neutral, PublicKeyToken=null | AppDomainAssemblyResolveEvent | AssemblyNotFound | |

| Event Name | AssemblyName | Stage | Result |
|------------------------------------|---|-----------------------------------|------------------|
| AssemblyLoader/ResolutionAttempted | MyLibrary, Culture=neutral, PublicKeyToken=null | FindInLoadContext | AssemblyNotFound |
| AssemblyLoader/ResolutionAttempted | MyLibrary, Culture=neutral, PublicKeyToken=null | ApplicationAssemblies | AssemblyNotFound |
| AssemblyLoader/ResolutionAttempted | MyLibrary, Culture=neutral, PublicKeyToken=null | AssemblyLoadContextResolvingEvent | AssemblyNotFound |
| AssemblyLoader/ResolutionAttempted | MyLibrary, Culture=neutral, PublicKeyToken=null | AppDomainAssemblyResolveEvent | AssemblyNotFound |

The events above indicate that the assembly loader attempted to resolve the assembly by looking in the current load context, running the default probing logic for managed application assemblies, invoking handlers for the `AssemblyLoadContext.Resolving` event, and invoking handlers for the `AppDomain.AssemblyResolve`. For all of these steps, the assembly was not found.

Extension points

To see which extension points were invoked, filter the view to the `AssemblyLoadContextResolvingHandlerInvoked` and `AppDomainAssemblyResolveHandlerInvoked` under `Microsoft-Windows-DotNETRuntime/AssemblyLoader` using the event list on the left. Add the columns `AssemblyName` and `HandlerName` to the view. Filter to events with the activity ID from the `Start`/`Stop` pair.

| Text Filter: | //1/2/ | Columns To Display: | Cols | AssemblyName HandlerName * |
|---|---|--------------------------------|------|----------------------------|
| Histogram: | | | | |
| Event Name | AssemblyName | HandlerName | | |
| Microsoft-Windows-DotNETRuntime/AssemblyLoader/AssemblyLoadContextResolvingHandlerInvoked | MyLibrary, Culture=neutral, PublicKeyToken=null | OnAssemblyLoadContextResolving | | |
| Microsoft-Windows-DotNETRuntime/AssemblyLoader/AppDomainAssemblyResolveHandlerInvoked | MyLibrary, Culture=neutral, PublicKeyToken=null | OnAppDomainAssemblyResolve | | |

| Event Name | AssemblyName | HandlerName |
|---|---|--------------------------------|
| AssemblyLoader/AssemblyLoadContextResolvingHandlerInvoked | MyLibrary, Culture=neutral, PublicKeyToken=null | OnAssemblyLoadContextResolving |
| AssemblyLoader/AppDomainAssemblyResolveHandlerInvoked | MyLibrary, Culture=neutral, PublicKeyToken=null | OnAppDomainAssemblyResolve |

The events above indicate that a handler named `OnAssemblyLoadContextResolving` was invoked for the `AssemblyLoadContext.Resolving` event and a handler named `OnAppDomainAssemblyResolve` was invoked for the `AppDomain.AssemblyResolve` event.

Collect another trace

Run the application with arguments such that its handler for the `AssemblyLoadContext.Resolving` event will load the `MyLibrary` assembly.

| Console | | | | |
|--|--|--|--|--|
| AssemblyLoading /d default alc-resolving | | | | |

Collect and open another `.nettrace` file using the [steps from above](#).

Filter to the `Start` and `Stop` events for `MyLibrary` again. You should see a `Start/Stop` pair with another `Start/Stop` between them. The inner load operation represents the load triggered by the handler for `AssemblyLoadContext.Resolving` when it called `AssemblyLoadContext.LoadFromAssemblyPath`. This time, you should see `Success=True` on the `Stop` event, indicating the load operation succeeded. The `ResultAssemblyPath` field shows the path of the resulting assembly.

| Event Name | AssemblyName | ActivityID | Success | ResultAssemblyPath |
|--|--|------------|---------|---|
| Microsoft-Windows-DotNETRuntime/AssemblyLoader/Start | MyLibrary, Culture=neutral, PublicKeyToken=null | //1/2/ | | |
| Microsoft-Windows-DotNETRuntime/AssemblyLoader/Start | MyLibrary, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null | //1/2/1/ | | |
| Microsoft-Windows-DotNETRuntime/AssemblyLoader/Stop | MyLibrary, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null | //1/2/1/ | True | C:\src\AssemblyLoading\bin\Debug\net5.0\MyLibrary.dll |
| Microsoft-Windows-DotNETRuntime/AssemblyLoader/Stop | MyLibrary, Culture=neutral, PublicKeyToken=null | //1/2/ | True | C:\src\AssemblyLoading\bin\Debug\net5.0\MyLibrary.dll |

| Event Name | AssemblyName | ActivityID | Success | ResultAssemblyPath |
|----------------------|--|------------|---------|---|
| AssemblyLoader/Start | MyLibrary, Culture=neutral, PublicKeyToken=null | //1/2/ | | |
| AssemblyLoader/Start | MyLibrary, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null | //1/2/1/ | | |
| AssemblyLoader/Stop | MyLibrary, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null | //1/2/1/ | True | C:\src\AssemblyLoading\bin\Debug\net5.0\MyLibrary.dll |
| AssemblyLoader/Stop | MyLibrary, Culture=neutral, PublicKeyToken=null | //1/2/ | True | C:\src\AssemblyLoading\bin\Debug\net5.0\MyLibrary.dll |

We can then look at the `ResolutionAttempted` events with the activity ID from the outer load to determine the step at which the assembly was successfully resolved. This time, the events will show that the `AssemblyLoadContextResolvingEvent` stage was successful. The `ResultAssemblyPath` field shows the path of the resulting assembly.

| Event Name | AssemblyName | Stage | Result | ResultAssemblyPath |
|--|----------------------------|-----------------------------------|------------------|---|
| Microsoft-Windows-DotNETRuntime/AssemblyLoader/ResolutionAttempted | MyLibrary, Culture=neutral | FindInLoadContext | AssemblyNotFound | |
| Microsoft-Windows-DotNETRuntime/AssemblyLoader/ResolutionAttempted | MyLibrary, Culture=neutral | ApplicationAssemblies | AssemblyNotFound | |
| Microsoft-Windows-DotNETRuntime/AssemblyLoader/ResolutionAttempted | MyLibrary, Culture=neutral | AssemblyLoadContextResolvingEvent | Success | C:\src\AssemblyLoading\bin\Debug\net5.0\MyLibrary.dll |

| Event Name | AssemblyName | Stage | Result | ResultAssemblyPath |
|------------------------------------|---|-----------------------------------|------------------|---|
| AssemblyLoader/ResolutionAttempted | MyLibrary, Culture=neutral, PublicKeyToken=null | FindInLoadContext | AssemblyNotFound | |
| AssemblyLoader/ResolutionAttempted | MyLibrary, Culture=neutral, PublicKeyToken=null | ApplicationAssemblies | AssemblyNotFound | |
| AssemblyLoader/ResolutionAttempted | MyLibrary, Culture=neutral, PublicKeyToken=null | AssemblyLoadContextResolvingEvent | Success | C:\src\AssemblyLoading\bin\Debug\net5.0\MyLibrary.dll |

Looking at `AssemblyLoadContextResolvingHandlerInvoked` events will show that the handler named `OnAssemblyLoadContextResolving` was invoked. The `ResultAssemblyPath` field shows the path of the assembly returned by the handler.

| Event Name | AssemblyName | HandlerName | ResultAssemblyPath |
|---|----------------------------|--------------------------------|---|
| Microsoft-Windows-DotNETRuntime/AssemblyLoader/AssemblyLoadContextResolvingHandlerInvoked | MyLibrary, Culture=neutral | OnAssemblyLoadContextResolving | C:\src\AssemblyLoading\bin\Debug\net5.0\MyLibrary.dll |

| Event Name | AssemblyName | HandlerName | ResultAssemblyPath |
|---|---|--------------------------------|---|
| AssemblyLoader/AssemblyLoadContextResolvingHandlerInvoked | MyLibrary, Culture=neutral, PublicKeyToken=null | OnAssemblyLoadContextResolving | C:\src\AssemblyLoading\bin\Debug\net5.0\MyLibrary.dll |

Note that there is no longer a `ResolutionAttempted` event with the `AppDomainAssemblyResolveEvent` stage or any `AppDomainAssemblyResolveHandlerInvoked` events, as the assembly was successfully loaded before reaching the step of the loading algorithm that raises the `AppDomain.AssemblyResolve` event.

See also

- [Assembly loader events](#)
- [dotnet-trace](#)
- [PerfView](#) ↗

🔗 Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



.NET feedback

The .NET documentation is open source. Provide feedback here.

🔗 [Open a documentation issue](#)

🔗 [Provide product feedback](#)

Create a .NET Core application with plugins

Article • 02/04/2022

This tutorial shows you how to create a custom [AssemblyLoadContext](#) to load plugins. An [AssemblyDependencyResolver](#) is used to resolve the dependencies of the plugin. The tutorial correctly isolates the plugin's dependencies from the hosting application. You'll learn how to:

- Structure a project to support plugins.
- Create a custom [AssemblyLoadContext](#) to load each plugin.
- Use the [System.Runtime.Loader.AssemblyDependencyResolver](#) type to allow plugins to have dependencies.
- Author plugins that can be easily deployed by just copying the build artifacts.

Prerequisites

- Install the [.NET 5 SDK](#) or a newer version.

ⓘ Note

The sample code targets .NET 5, but all the features it uses were introduced in .NET Core 3.0 and are available in all .NET releases since then.

Create the application

The first step is to create the application:

1. Create a new folder, and in that folder run the following command:

```
.NET CLI  
dotnet new console -o AppWithPlugin
```

2. To make building the project easier, create a Visual Studio solution file in the same folder. Run the following command:

```
.NET CLI
```

```
dotnet new sln
```

- Run the following command to add the app project to the solution:

.NET CLI

```
dotnet sln add AppWithPlugin/AppWithPlugin.csproj
```

Now we can fill in the skeleton of our application. Replace the code in the *AppWithPlugin/Program.cs* file with the following code:

C#

```
using PluginBase;
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Reflection;

namespace AppWithPlugin
{
    class Program
    {
        static void Main(string[] args)
        {
            try
            {
                if (args.Length == 1 && args[0] == "/d")
                {
                    Console.WriteLine("Waiting for any key...");
                    Console.ReadLine();
                }

                // Load commands from plugins.

                if (args.Length == 0)
                {
                    Console.WriteLine("Commands: ");
                    // Output the loaded commands.
                }
                else
                {
                    foreach (string commandName in args)
                    {
                        Console.WriteLine($"-- {commandName} --");

                        // Execute the command with the name passed as an
                        argument.
                    }
                }
            }
        }
    }
}
```

```
        Console.WriteLine();
    }
}
catch (Exception ex)
{
    Console.WriteLine(ex);
}
}
}
```

Create the plugin interfaces

The next step in building an app with plugins is defining the interface the plugins need to implement. We suggest that you make a class library that contains any types that you plan to use for communicating between your app and plugins. This division allows you to publish your plugin interface as a package without having to ship your full application.

In the root folder of the project, run `dotnet new classlib -o PluginBase`. Also, run `dotnet sln add PluginBase/PluginBase.csproj` to add the project to the solution file. Delete the `PluginBase/Class1.cs` file, and create a new file in the `PluginBase` folder named `ICommand.cs` with the following interface definition:

```
C#  
  
namespace PluginBase  
{  
    public interface ICommand  
    {  
        string Name { get; }  
        string Description { get; }  
  
        int Execute();  
    }  
}
```

This `ICommand` interface is the interface that all of the plugins will implement.

Now that the `ICommand` interface is defined, the application project can be filled in a little more. Add a reference from the `AppWithPlugin` project to the `PluginBase` project with the `dotnet add AppWithPlugin/AppWithPlugin.csproj reference PluginBase/PluginBase.csproj` command from the root folder.

Replace the `// Load commands from plugins` comment with the following code snippet to enable it to load plugins from given file paths:

```
C#  
  
string[] pluginPaths = new string[]  
{  
    // Paths to plugins to load.  
};  
  
IEnumerable< ICommand> commands = pluginPaths.SelectMany(pluginPath =>  
{  
    Assembly pluginAssembly = LoadPlugin(pluginPath);  
    return CreateCommands(pluginAssembly);  
}).ToList();
```

Then replace the `// Output the loaded commands` comment with the following code snippet:

```
C#  
  
foreach ( ICommand command in commands)  
{  
    Console.WriteLine($"{command.Name}\t - {command.Description}");  
}
```

Replace the `// Execute the command with the name passed as an argument` comment with the following snippet:

```
C#  
  
ICommand command = commands.FirstOrDefault(c => c.Name == commandName);  
if (command == null)  
{  
    Console.WriteLine("No such command is known.");  
    return;  
}  
  
command.Execute();
```

And finally, add static methods to the `Program` class named `LoadPlugin` and `CreateCommands`, as shown here:

```
C#  
  
static Assembly LoadPlugin(string relativePath)  
{
```

```

        throw new NotImplementedException();
    }

    static IEnumerable< ICommand> CreateCommands(Assembly assembly)
    {
        int count = 0;

        foreach (Type type in assembly.GetTypes())
        {
            if (typeof(ICommand).IsAssignableFrom(type))
            {
                ICommand result = Activator.CreateInstance(type) as ICommand;
                if (result != null)
                {
                    count++;
                    yield return result;
                }
            }
        }

        if (count == 0)
        {
            string availableTypes = string.Join(",",
assembly.GetTypes().Select(t => t.FullName));
            throw new ApplicationException(
                $"Can't find any type which implements ICommand in {assembly}
from {assembly.Location}.\\n" +
                $"Available types: {availableTypes}");
        }
    }
}

```

Load plugins

Now the application can correctly load and instantiate commands from loaded plugin assemblies, but it's still unable to load the plugin assemblies. Create a file named *PluginLoadContext.cs* in the *AppWithPlugin* folder with the following contents:

C#

```

using System;
using System.Reflection;
using System.Runtime.Loader;

namespace AppWithPlugin
{
    class PluginLoadContext : AssemblyLoadContext
    {
        private AssemblyDependencyResolver _resolver;

        public PluginLoadContext(string pluginPath)
        {

```

```

        _resolver = new AssemblyDependencyResolver(pluginPath);
    }

    protected override Assembly Load(AssemblyName assemblyName)
    {
        string assemblyPath =
_resolver.ResolveAssemblyToPath(assemblyName);
        if (assemblyPath != null)
        {
            return LoadFromAssemblyPath(assemblyPath);
        }

        return null;
    }

    protected override IntPtr LoadUnmanagedDll(string unmanagedDllName)
    {
        string libraryPath =
_resolver.ResolveUnmanagedDllToPath(unmanagedDllName);
        if (libraryPath != null)
        {
            return LoadUnmanagedDllFromPath(libraryPath);
        }

        return IntPtr.Zero;
    }
}

```

The `PluginLoadContext` type derives from `AssemblyLoadContext`. The `AssemblyLoadContext` type is a special type in the runtime that allows developers to isolate loaded assemblies into different groups to ensure that assembly versions don't conflict. Additionally, a custom `AssemblyLoadContext` can choose different paths to load assemblies from and override the default behavior. The `PluginLoadContext` uses an instance of the `AssemblyDependencyResolver` type introduced in .NET Core 3.0 to resolve assembly names to paths. The `AssemblyDependencyResolver` object is constructed with the path to a .NET class library. It resolves assemblies and native libraries to their relative paths based on the `.deps.json` file for the class library whose path was passed to the `AssemblyDependencyResolver` constructor. The custom `AssemblyLoadContext` enables plugins to have their own dependencies, and the `AssemblyDependencyResolver` makes it easy to correctly load the dependencies.

Now that the `AppWithPlugin` project has the `PluginLoadContext` type, update the `Program.LoadPlugin` method with the following body:

C#

```

static Assembly LoadPlugin(string relativePath)
{
    // Navigate up to the solution root
    string root = Path.GetFullPath(Path.Combine(
        Path.GetDirectoryName(
            Path.GetDirectoryName(
                Path.GetDirectoryName(
                    Path.GetDirectoryName(
                        Path.GetDirectoryName(
                            Path.GetDirectoryName(
                                Path.GetDirectoryName(
                                    Path.GetDirectoryName(
                                        Path.GetDirectoryName(
                                            Path.GetDirectoryName(
                                                Path.GetDirectoryName(
                                                    typeof(Program).Assembly.Location))))))))));
    string pluginLocation = Path.GetFullPath(Path.Combine(root,
relativePath.Replace('\\', Path.DirectorySeparatorChar)));
    Console.WriteLine($"Loading commands from: {pluginLocation}");
    PluginLoadContext loadContext = new PluginLoadContext(pluginLocation);
    return loadContext.LoadFromAssemblyName(new
AssemblyName(Path.GetFileNameWithoutExtension(pluginLocation)));
}

```

By using a different `PluginLoadContext` instance for each plugin, the plugins can have different or even conflicting dependencies without issue.

Simple plugin with no dependencies

Back in the root folder, do the following:

1. Run the following command to create a new class library project named

`HelloPlugin:`

```

.NET CLI
dotnet new classlib -o HelloPlugin

```

2. Run the following command to add the project to the `AppWithPlugin` solution:

```

.NET CLI
dotnet sln add HelloPlugin/HelloPlugin.csproj

```

3. Replace the `HelloPlugin/Class1.cs` file with a file named `HelloCommand.cs` with the following contents:

C#

```

using PluginBase;
using System;

```

```
namespace HelloPlugin
{
    public class HelloCommand : ICommand
    {
        public string Name { get => "hello"; }
        public string Description { get => "Displays hello message."; }

        public int Execute()
        {
            Console.WriteLine("Hello !!!");
            return 0;
        }
    }
}
```

Now, open the *HelloPlugin.csproj* file. It should look similar to the following:

XML

```
<Project Sdk="Microsoft.NET.Sdk">

<PropertyGroup>
    <TargetFramework>net5.0</TargetFramework>
</PropertyGroup>

</Project>
```

In between the `<PropertyGroup>` tags, add the following element:

XML

```
<EnableDynamicLoading>true</EnableDynamicLoading>
```

The `<EnableDynamicLoading>true</EnableDynamicLoading>` prepares the project so that it can be used as a plugin. Among other things, this will copy all of its dependencies to the output of the project. For more details see [EnableDynamicLoading](#).

In between the `<Project>` tags, add the following elements:

XML

```
<ItemGroup>
    <ProjectReference Include="..\PluginBase\PluginBase.csproj">
        <Private>false</Private>
        <ExcludeAssets>runtime</ExcludeAssets>
```

```
</ProjectReference>  
</ItemGroup>
```

The `<Private>false</Private>` element is important. This tells MSBuild to not copy `PluginBase.dll` to the output directory for `HelloPlugin`. If the `PluginBase.dll` assembly is present in the output directory, `PluginLoadContext` will find the assembly there and load it when it loads the `HelloPlugin.dll` assembly. At this point, the `HelloPlugin.HelloCommand` type will implement the `ICommand` interface from the `PluginBase.dll` in the output directory of the `HelloPlugin` project, not the `ICommand` interface that is loaded into the default load context. Since the runtime sees these two types as different types from different assemblies, the `AppWithPlugin.Program.CreateCommands` method won't find the commands. As a result, the `<Private>false</Private>` metadata is required for the reference to the assembly containing the plugin interfaces.

Similarly, the `<ExcludeAssets>runtime</ExcludeAssets>` element is also important if the `PluginBase` references other packages. This setting has the same effect as `<Private>false</Private>` but works on package references that the `PluginBase` project or one of its dependencies may include.

Now that the `HelloPlugin` project is complete, you should update the `AppWithPlugin` project to know where the `HelloPlugin` plugin can be found. After the `// Paths to plugins to load` comment, add `["@HelloPlugin\bin\Debug\net5.0\HelloPlugin.dll"]` (this path could be different based on the .NET Core version you use) as an element of the `pluginPaths` array.

Plugin with library dependencies

Almost all plugins are more complex than a simple "Hello World", and many plugins have dependencies on other libraries. The `JsonPlugin` and `OldJsonPlugin` projects in the sample show two examples of plugins with NuGet package dependencies on `Newtonsoft.Json`. Because of this, all plugin projects should add `<EnableDynamicLoading>true</EnableDynamicLoading>` to the project properties so that they copy all of their dependencies to the output of `dotnet build`. Publishing the class library with `dotnet publish` will also copy all of its dependencies to the publish output.

Other examples in the sample

The complete source code for this tutorial can be found in [the dotnet/samples repository](#). The completed sample includes a few other examples of

`AssemblyDependencyResolver` behavior. For example, the `AssemblyDependencyResolver` object can also resolve native libraries as well as localized satellite assemblies included in NuGet packages. The `UVPlugin` and `FrenchPlugin` in the samples repository demonstrate these scenarios.

Reference a plugin interface from a NuGet package

Let's say that there is an app A that has a plugin interface defined in the NuGet package named `A.PluginBase`. How do you reference the package correctly in your plugin project? For project references, using the `<Private>false</Private>` metadata on the `ProjectReference` element in the project file prevented the dll from being copied to the output.

To correctly reference the `A.PluginBase` package, you want to change the `<PackageReference>` element in the project file to the following:

XML

```
<PackageReference Include="A.PluginBase" Version="1.0.0">
    <ExcludeAssets>runtime</ExcludeAssets>
</PackageReference>
```

This prevents the `A.PluginBase` assemblies from being copied to the output directory of your plugin and ensures that your plugin will use A's version of `A.PluginBase`.

Plugin target framework recommendations

Because plugin dependency loading uses the `.deps.json` file, there is a gotcha related to the plugin's target framework. Specifically, your plugins should target a runtime, such as .NET 5, instead of a version of .NET Standard. The `.deps.json` file is generated based on which framework the project targets, and since many .NET Standard-compatible packages ship reference assemblies for building against .NET Standard and implementation assemblies for specific runtimes, the `.deps.json` may not correctly see implementation assemblies, or it may grab the .NET Standard version of an assembly instead of the .NET Core version you expect.

Plugin framework references

Currently, plugins can't introduce new frameworks into the process. For example, you can't load a plugin that uses the `Microsoft.AspNetCore.App` framework into an application that only uses the root `Microsoft.NETCore.App` framework. The host application must declare references to all frameworks needed by plugins.

 **Collaborate with us on GitHub**

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

How to use and debug assembly unloadability in .NET

Article • 11/13/2023

.NET (Core) introduced the ability to load and later unload a set of assemblies. In .NET Framework, custom app domains were used for this purpose, but .NET (Core) only supports a single default app domain.

Unloadability is supported through [AssemblyLoadContext](#). You can load a set of assemblies into a collectible `AssemblyLoadContext`, execute methods in them or just inspect them using reflection, and finally unload the `AssemblyLoadContext`. That unloads the assemblies loaded into the `AssemblyLoadContext`.

There's one noteworthy difference between the unloading using `AssemblyLoadContext` and using AppDomains. With AppDomains, the unloading is forced. At unload time, all threads running in the target AppDomain are aborted, managed COM objects created in the target AppDomain are destroyed, and so on. With `AssemblyLoadContext`, the unload is "cooperative". Calling the [AssemblyLoadContext.Unload](#) method just initiates the unloading. The unloading finishes after:

- No threads have methods from the assemblies loaded into the `AssemblyLoadContext` on their call stacks.
- None of the types from the assemblies loaded into the `AssemblyLoadContext`, instances of those types, and the assemblies themselves are referenced by:
 - References outside of the `AssemblyLoadContext`, except for weak references ([WeakReference](#) or [WeakReference<T>](#)).
 - Strong garbage collector (GC) handles ([GCHandleType.Normal](#) or [GCHandleType.Pinned](#)) from both inside and outside of the `AssemblyLoadContext`.

Use collectible AssemblyLoadContext

This section contains a detailed step-by-step tutorial that shows a simple way to load a .NET (Core) application into a collectible `AssemblyLoadContext`, execute its entry point, and then unload it. You can find a complete sample at <https://github.com/dotnet/samples/tree/main/core/tutorials/Unloading>.

Create a collectible AssemblyLoadContext

Derive your class from the [AssemblyLoadContext](#) and override its [AssemblyLoadContext.Load](#) method. That method resolves references to all assemblies that are dependencies of assemblies loaded into that [AssemblyLoadContext](#).

The following code is an example of the simplest custom [AssemblyLoadContext](#):

C#

```
class TestAssemblyLoadContext : AssemblyLoadContext
{
    public TestAssemblyLoadContext() : base(isCollectible: true)
    {

    }

    protected override Assembly? Load(AssemblyName name)
    {
        return null;
    }
}
```

As you can see, the `Load` method returns `null`. That means that all the dependency assemblies are loaded into the default context, and the new context contains only the assemblies explicitly loaded into it.

If you want to load some or all of the dependencies into the [AssemblyLoadContext](#) too, you can use the [AssemblyDependencyResolver](#) in the `Load` method. The [AssemblyDependencyResolver](#) resolves the assembly names to absolute assembly file paths. The resolver uses the `.deps.json` file and assembly files in the directory of the main assembly loaded into the context.

C#

```
using System.Reflection;
using System.Runtime.Loader;

namespace complex
{
    class TestAssemblyLoadContext : AssemblyLoadContext
    {
        private AssemblyDependencyResolver _resolver;

        public TestAssemblyLoadContext(string mainAssemblyToLoadPath) :
base(isCollectible: true)
        {
            _resolver = new
AssemblyDependencyResolver(mainAssemblyToLoadPath);
        }

        protected override Assembly? Load(AssemblyName name)
```

```
        {
            string? assemblyPath = _resolver.ResolveAssemblyToPath(name);
            if (assemblyPath != null)
            {
                return LoadFromAssemblyPath(assemblyPath);
            }

            return null;
        }
    }
}
```

Use a custom collectible AssemblyLoadContext

This section assumes the simpler version of the `TestAssemblyLoadContext` is being used.

You can create an instance of the custom `AssemblyLoadContext` and load an assembly into it as follows:

C#

```
var alc = new TestAssemblyLoadContext();
Assembly a = alc.LoadFromAssemblyPath(assemblyPath);
```

For each of the assemblies referenced by the loaded assembly, the `TestAssemblyLoadContext.Load` method is called so that the `TestAssemblyLoadContext` can decide where to get the assembly from. In this case, it returns `null` to indicate that it should be loaded into the default context from locations that the runtime uses to load assemblies by default.

Now that an assembly was loaded, you can execute a method from it. Run the `Main` method:

C#

```
var args = new object[1] {new string[] {"Hello"}};
_ = a.EntryPoint?.Invoke(null, args);
```

After the `Main` method returns, you can initiate unloading by either calling the `Unload` method on the custom `AssemblyLoadContext` or removing the reference you have to the `AssemblyLoadContext`:

C#

```
alc.Unload();
```

This is sufficient to unload the test assembly. Next, you'll put all of this into a separate noninlineable method to ensure that the `TestAssemblyLoadContext`, `Assembly`, and `MethodInfo` (the `Assembly.EntryPoint`) can't be kept alive by stack slot references (real- or JIT-introduced locals). That could keep the `TestAssemblyLoadContext` alive and prevent the unload.

Also, return a weak reference to the `AssemblyLoadContext` so that you can use it later to detect unload completion.

C#

```
[MethodImpl(MethodImplOptions.NoInlining)]
static void ExecuteAndUnload(string assemblyPath, out WeakReference
alcWeakRef)
{
    var alc = new TestAssemblyLoadContext();
    Assembly a = alc.LoadFromAssemblyPath(assemblyPath);

    alcWeakRef = new WeakReference(alc, trackResurrection: true);

    var args = new object[1] {new string[] {"Hello"}};
    _ = a.EntryPoint?.Invoke(null, args);

    alc.Unload();
}
```

Now you can run this function to load, execute, and unload the assembly.

C#

```
WeakReference testAlcWeakRef;
ExecuteAndUnload("absolute/path/to/your/assembly", out testAlcWeakRef);
```

However, the unload doesn't complete immediately. As previously mentioned, it relies on the garbage collector to collect all the objects from the test assembly. In many cases, it isn't necessary to wait for the unload completion. However, there are cases where it's useful to know that the unload has finished. For example, you might want to delete the assembly file that was loaded into the custom `AssemblyLoadContext` from disk. In such a case, the following code snippet can be used. It triggers garbage collection and waits for pending finalizers in a loop until the weak reference to the custom `AssemblyLoadContext` is set to `null`, indicating the target object was collected. In most cases, just one pass through the loop is required. However, for more complex cases where objects created

by the code running in the `AssemblyLoadContext` have finalizers, more passes might be needed.

C#

```
for (int i = 0; testAlcWeakRef.IsAlive && (i < 10); i++)
{
    GC.Collect();
    GC.WaitForPendingFinalizers();
}
```

The Unloading event

In some cases, it might be necessary for the code loaded into a custom `AssemblyLoadContext` to perform some cleanup when the unloading is initiated. For example, it might need to stop threads or clean up strong GC handles. The `Unloading` event can be used in such cases. You can hook a handler that performs the necessary cleanup to this event.

Troubleshoot unloadability issues

Due to the cooperative nature of the unloading, it's easy to forget about references that might be keeping the stuff in a collectible `AssemblyLoadContext` alive and preventing unload. Here's a summary of entities (some of them nonobvious) that can hold the references:

- Regular references held from outside of the collectible `AssemblyLoadContext` that are stored in a stack slot or a processor register (method locals, either explicitly created by the user code or implicitly by the just-in-time (JIT) compiler), a static variable, or a strong (pinning) GC handle, and transitively pointing to:
 - An assembly loaded into the collectible `AssemblyLoadContext`.
 - A type from such an assembly.
 - An instance of a type from such an assembly.
- Threads running code from an assembly loaded into the collectible `AssemblyLoadContext`.
- Instances of custom, noncollectible `AssemblyLoadContext` types created inside of the collectible `AssemblyLoadContext`.
- Pending `RegisteredWaitHandle` instances with callbacks set to methods in the custom `AssemblyLoadContext`.

Tip

Object references that are stored in stack slots or processor registers and that could prevent unloading of an `AssemblyLoadContext` can occur in the following situations:

- When function call results are passed directly to another function, even though there is no user-created local variable.
- When the JIT compiler keeps a reference to an object that was available at some point in a method.

Debug unloading issues

Debugging issues with unloading can be tedious. You can get into situations where you don't know what can be holding an `AssemblyLoadContext` alive, but the unload fails. The best tool to help with that is WinDbg (or LLDB on Unix) with the SOS plugin. You need to find what's keeping a `LoaderAllocator` that belongs to the specific `AssemblyLoadContext` alive. The SOS plugin lets you look at GC heap objects, their hierarchies, and roots.

To load the SOS plugin into the debugger, enter one of the following commands in the debugger command line.

In WinDbg (if it's not already loaded):

```
Console  
.loadby sos coreclr
```

In LLDB:

```
Console  
plugin load /path/to/libssosplugin.so
```

Now you'll debug an example program that has problems with unloading. The source code is available in the [Example source code](#) section. When you run it under WinDbg, the program breaks into the debugger right after attempting to check for the unload success. You can then start looking for the culprits.

 **Tip**

If you debug using LLDB on Unix, the SOS commands in the following examples don't have the ! in front of them.

Console

```
!dumpheap -type LoaderAllocator
```

This command dumps all objects with a type name containing `LoaderAllocator` that are in the GC heap. Here's an example:

Console

| Address | MT | Size |
|------------------|------------------|------|
| 000002b78000ce40 | 00007ffadc93a288 | 48 |
| 000002b78000ceb0 | 00007ffadc93a218 | 24 |

Statistics:

| MT | Count | TotalSize | Class | Name |
|--|-------|-----------|-----------------------------------|------|
| 00007ffadc93a218 | 1 | 24 | | |
| System.Reflection.LoaderAllocatorScout | | | | |
| 00007ffadc93a288 | 1 | 48 | System.Reflection.LoaderAllocator | |
| Total | 2 | objects | | |

In the "Statistics:" part, check the `MT` (`MethodTable`) that belongs to the `System.Reflection.LoaderAllocator`, which is the object you care about. Then, in the list at the beginning, find the entry with `MT` that matches that one, and get the address of the object itself. In this case, it's "000002b78000ce40".

Now that you know the address of the `LoaderAllocator` object, you can use another command to find its GC roots:

Console

```
!gcroot 0x000002b78000ce40
```

This command dumps the chain of object references that lead to the `LoaderAllocator` instance. The list starts with the root, which is the entity that keeps the `LoaderAllocator` alive and thus is the core of the problem. The root can be a stack slot, a processor register, a GC handle, or a static variable.

Here's an example of the output of the `gcroot` command:

Console

```

Thread 4ac:
  000000cf9499dd20 00007ffa7d0236bc example.Program.Main(System.String[])
[E:\unloadability\example\Program.cs @ 70]
  rbp-20: 000000cf9499dd90
    -> 000002b78000d328 System.Reflection.RuntimeMethodInfo
    -> 000002b78000d1f8 System.RuntimeType+RuntimeTypeCache
    -> 000002b78000d1d0 System.RuntimeType
    -> 000002b78000ce40 System.Reflection.LoaderAllocator

HandleTable:
  000002b7f8a81198 (strong handle)
  -> 000002b78000d948 test.Test
  -> 000002b78000ce40 System.Reflection.LoaderAllocator

  000002b7f8a815f8 (pinned handle)
  -> 000002b790001038 System.Object[]
  -> 000002b78000d390 example.TestInfo
  -> 000002b78000d328 System.Reflection.RuntimeMethodInfo
  -> 000002b78000d1f8 System.RuntimeType+RuntimeTypeCache
  -> 000002b78000d1d0 System.RuntimeType
  -> 000002b78000ce40 System.Reflection.LoaderAllocator

Found 3 roots.

```

The next step is to figure out where the root is located so you can fix it. The easiest case is when the root is a stack slot or a processor register. In that case, the `gcroot` shows the name of the function whose frame contains the root and the thread executing that function. The difficult case is when the root is a static variable or a GC handle.

In the previous example, the first root is a local of type `System.Reflection.RuntimeMethodInfo` stored in the frame of the function `example.Program.Main(System.String[])` at address `rbp-20` (`rbp` is the processor register `rbp` and `-20` is a hexadecimal offset from that register).

The second root is a normal (strong) `GCHandle` that holds a reference to an instance of the `test.Test` class.

The third root is a pinned `GCHandle`. This one is actually a static variable, but unfortunately, there's no way to tell. Statics for reference types are stored in a managed object array in internal runtime structures.

Another case that can prevent unloading of an `AssemblyLoadContext` is when a thread has a frame of a method from an assembly loaded into the `AssemblyLoadContext` on its stack. You can check that by dumping managed call stacks of all threads:

Console

```
~*e !clrstack
```

The command means "apply to all threads the `!clrstack` command". The following is the output of that command for the example. Unfortunately, LLDB on Unix doesn't have any way to apply a command to all threads, so you must manually switch threads and repeat the `clrstack` command. Ignore all threads where the debugger says "Unable to walk the managed stack".

Console

```
OS Thread Id: 0x6ba8 (0)
    Child SP          IP Call Site
0000001fc697d5c8 00007ffb50d9de12 [HelperMethodFrame: 0000001fc697d5c8]
System.Diagnostics.Debugger.BreakInternal()
0000001fc697d6d0 00007ffa864765fa System.Diagnostics.Debugger.Break()
0000001fc697d700 00007ffa864736bc example.Program.Main(System.String[])
[E:\unloadability\example\Program.cs @ 70]
0000001fc697d998 00007ffae5fdc1e3 [GCFrame: 0000001fc697d998]
0000001fc697df28 00007ffae5fdc1e3 [GCFrame: 0000001fc697df28]
OS Thread Id: 0x2ae4 (1)
Unable to walk the managed stack. The current thread is likely not a
managed thread. You can run !threads to get a list of managed threads in
the process
Failed to start stack walk: 80070057
OS Thread Id: 0x61a4 (2)
Unable to walk the managed stack. The current thread is likely not a
managed thread. You can run !threads to get a list of managed threads in
the process
Failed to start stack walk: 80070057
OS Thread Id: 0x7fdc (3)
Unable to walk the managed stack. The current thread is likely not a
managed thread. You can run !threads to get a list of managed threads in
the process
Failed to start stack walk: 80070057
OS Thread Id: 0x5390 (4)
Unable to walk the managed stack. The current thread is likely not a
managed thread. You can run !threads to get a list of managed threads in
the process
Failed to start stack walk: 80070057
OS Thread Id: 0x5ec8 (5)
    Child SP          IP Call Site
0000001fc70ff6e0 00007ffb5437f6e4 [DebuggerU2MCatchHandlerFrame:
0000001fc70ff6e0]
OS Thread Id: 0x4624 (6)
    Child SP          IP Call Site
GetFrameContext failed: 1
0000000000000000 0000000000000000
OS Thread Id: 0x60bc (7)
    Child SP          IP Call Site
0000001fc727f158 00007ffb5437fce4 [HelperMethodFrame: 0000001fc727f158]
System.Threading.Thread.SleepInternal(Int32)
```

```
0000001fc727f260 00007ffb37ea7c2b System.Threading.Thread.Sleep(Int32)
0000001fc727f290 00007ffa865005b3 test.Program.ThreadProc()
[E:\unloadability\test\Program.cs @ 17]
0000001fc727f2c0 00007ffb37ea6a5b
System.Threading.Thread.ThreadMain_ThreadStart()
0000001fc727f2f0 00007ffadbc4cbe3
System.Threading.ExecutionContext.RunInternal(System.Threading.ExecutionCont
ext, System.Threading.ContextCallback, System.Object)
0000001fc727f568 00007ffae5fdc1e3 [GCFrame: 0000001fc727f568]
0000001fc727f7f0 00007ffae5fdc1e3 [DebuggerU2MCatchHandlerFrame:
0000001fc727f7f0]
```

As you can see, the last thread has `test.Program.ThreadProc()`. This is a function from the assembly loaded into the `AssemblyLoadContext`, and so it keeps the `AssemblyLoadContext` alive.

Example source code

The following code that contains unloadability issues is used in the previous debugging example.

Main testing program

```
C#  
  
using System;
using System.Reflection;
using System.Runtime.CompilerServices;
using System.Runtime.Loader;  
  
namespace example
{
    class TestAssemblyLoadContext : AssemblyLoadContext
    {
        public TestAssemblyLoadContext() : base(true)
        {
        }
        protected override Assembly? Load(AssemblyName name)
        {
            return null;
        }
    }

    class TestInfo
    {
        public TestInfo(MethodInfo? mi)
        {
            _entryPoint = mi;
        }
    }
}
```

```

        MethodInfo? _entryPoint;
    }

    class Program
    {
        static TestInfo? entryPoint;

        [MethodImpl(MethodImplOptions.NoInlining)]
        static int ExecuteAndUnload(string assemblyPath, out WeakReference testAlcWeakRef, out MethodInfo? testEntryPoint)
        {
            var alc = new TestAssemblyLoadContext();
            testAlcWeakRef = new WeakReference(alc);

            Assembly a = alc.LoadFromAssemblyPath(assemblyPath);
            if (a == null)
            {
                testEntryPoint = null;
                Console.WriteLine("Loading the test assembly failed");
                return -1;
            }

            var args = new object[1] {new string[] {"Hello"}};

            // Issue preventing unloading #1 - we keep MethodInfo of a
method
            // for an assembly loaded into the TestAssemblyLoadContext in a
static variable.
            entryPoint = new TestInfo(a.EntryPoint);
            testEntryPoint = a.EntryPoint;

            var oResult = a.EntryPoint?.Invoke(null, args);
            alc.Unload();
            return (oResult is int result) ? result : -1;
        }

        static void Main(string[] args)
        {
            WeakReference testAlcWeakRef;
            // Issue preventing unloading #2 - we keep MethodInfo of a
method for an assembly loaded into the TestAssemblyLoadContext in a local
variable
            MethodInfo? testEntryPoint;
            int result = ExecuteAndUnload(@"absolute/path/to/test.dll", out
testAlcWeakRef, out testEntryPoint);

            for (int i = 0; testAlcWeakRef.IsAlive && (i < 10); i++)
            {
                GC.Collect();
                GC.WaitForPendingFinalizers();
            }

            System.Diagnostics.Debugger.Break();
        }
    }
}

```

```
        Console.WriteLine($"Test completed, result={result}, entryPoint:  
{testEntryPoint} unload success: {!testAlcWeakRef.IsAlive}");  
    }  
}  
}
```

Program loaded into the TestAssemblyLoadContext

The following code represents the *test.dll* passed to the `ExecuteAndUnload` method in the main testing program.

C#

```
using System;  
using System.Runtime.InteropServices;  
using System.Threading;  
  
namespace test  
{  
    class Test  
    {  
    }  
  
    class Program  
    {  
        public static void ThreadProc()  
        {  
            // Issue preventing unloading #4 - a thread running method  
            // inside of the TestAssemblyLoadContext at the unload time  
            Thread.Sleep(Timeout.Infinite);  
        }  
  
        static GCHandle handle;  
        static int Main(string[] args)  
        {  
            // Issue preventing unloading #3 - normal GC handle  
            handle = GCHandle.Alloc(new Test());  
            Thread t = new Thread(new ThreadStart(ThreadProc));  
            t.IsBackground = true;  
            t.Start();  
            Console.WriteLine($"Hello from the test: args[0] = {args[0]}");  
  
            return 1;  
        }  
    }  
}
```

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

How .NET is versioned

Article • 04/27/2023

The [.NET Runtime](#) and the [.NET SDK](#) add new features at different frequencies. In general, the SDK is updated more frequently than the Runtime. This article explains the runtime and the SDK version numbers.

.NET releases a new major version every November. Even-numbered releases, such as .NET 6 or .NET 8, are long-term supported (LTS). LTS releases get free support and patches for three years. Odd-numbered releases are standard-term support. Standard-term support releases get free support and patches for 18 months.

Versioning details

The .NET Runtime has a major.minor.patch approach to versioning that follows [semantic versioning](#).

The .NET SDK, however, doesn't follow semantic versioning. The .NET SDK releases faster and its version numbers must communicate both the aligned runtime and the SDK's own minor and patch releases.

The first two positions of the .NET SDK version number match the .NET Runtime version it released with. Each version of the SDK can create applications for this runtime or any lower version.

The third position of the SDK version number communicates both the minor and patch number. The minor version is multiplied by 100. The final two digits represent the patch number. Minor version 1, patch version 2 would be represented as 102. For example, here's a possible sequence of runtime and SDK version numbers:

| Change | .NET Runtime | .NET SDK (*) | Notes |
|-----------------------|--------------|--------------|---|
| Initial release | 5.0.0 | 5.0.100 | Initial release. |
| SDK patch | 5.0.0 | 5.0.101 | Runtime didn't change with this SDK patch. SDK patch bumps last digit in SDK patch. |
| Runtime and SDK patch | 5.0.1 | 5.0.102 | Runtime patch bumps Runtime patch number. SDK patch bumps last digit in SDK patch. |
| SDK feature change | 5.0.1 | 5.0.200 | Runtime patch didn't change. New SDK feature bumps first digit in SDK patch. |

| Change | .NET Runtime | .NET SDK (*) | Notes |
|---------------|--------------|--------------|---|
| Runtime patch | 5.0.2 | 5.0.200 | Runtime patch bumps Runtime patch number. SDK doesn't change. |

From the preceding table you can see several policies:

- The Runtime and SDK share major and minor versions. The first two numbers for a given SDK and runtime should match. All the preceding examples are part of the .NET 5.0 release stream.
- The patch version of the runtime revs only when the runtime is updated. The SDK patch number doesn't update for a runtime patch.
- The patch version of the SDK updates only when the SDK is updated. It's possible that a runtime patch doesn't require an SDK patch.

NOTES:

- If the SDK has 10 feature updates before a runtime feature update, version numbers roll into the 1000 series. Version 5.0.1000 would follow version 5.0.900. This situation isn't expected to occur.
- 99 patch releases without a feature release won't occur. If a release approaches this number, it forces a feature release.

You can see more details in the initial proposal at the [dotnet/designs](#) repository.

Semantic versioning

The .NET *Runtime* roughly adheres to [Semantic Versioning \(SemVer\)](#), adopting the use of `MAJOR.MINOR.PATCH` versioning, using the various parts of the version number to describe the degree and type of change.

`MAJOR.MINOR.PATCH[-PRERELEASE-BUILDSNUMBER]`

The optional `PRERELEASE` and `BUILDSNUMBER` parts are never part of supported releases and only exist on nightly builds, local builds from source targets, and unsupported preview releases.

Understand runtime version number changes

- `MAJOR` is incremented once a year and may contain:
 - Significant changes in the product, or a new product direction.
 - API introduced breaking changes. There's a high bar to accepting breaking changes.
 - A newer `MAJOR` version of an existing dependency is adopted.

Major releases happen once a year, even-numbered versions are long-term supported (LTS) releases. The first LTS release using this versioning scheme is .NET 6. The latest non-LTS version is .NET 5.

- `MINOR` is incremented when:
 - Public API surface area is added.
 - A new behavior is added.
 - A newer `MINOR` version of an existing dependency is adopted.
 - A new dependency is introduced.
- `PATCH` is incremented when:
 - Bug fixes are made.
 - Support for a newer platform is added.
 - A newer `PATCH` version of an existing dependency is adopted.
 - Any other change doesn't fit one of the previous cases.

When there are multiple changes, the highest element affected by individual changes is incremented, and the following ones are reset to zero. For example, when `MAJOR` is incremented, `MINOR.PATCH` are reset to zero. When `MINOR` is incremented, `PATCH` is reset to zero while `MAJOR` remains the same.

Version numbers in file names

The files downloaded for .NET carry the version, for example, `dotnet-sdk-5.0.301-win10-x64.exe`.

Preview versions

Preview versions have a `-preview.[number].[build]` appended to the version number. For example, `6.0.0-preview.5.21302.13`.

Servicing versions

After a release goes out, the release branches generally stop producing daily builds and instead start producing servicing builds. Servicing versions have a `-servicing-[number]` appended to the version. For example, `5.0.1-servicing-006924`.

.NET runtime compatibility

The .NET runtime maintains a high level of compatibility between versions. .NET apps should, by and large, continue to work after upgrading to a new major .NET runtime version.

Each major .NET runtime version contains intentional, carefully vetted, and documented [breaking changes](#). The documented breaking changes aren't the only source of issues that can affect an app after upgrade. For example, a performance improvement in the .NET runtime (that's not considered a breaking change) can expose latent app threading bugs that cause the app to not work on that version. It's expected for large apps to require a few fixes after upgrading to a new .NET runtime major version.

By default, .NET apps are configured to run on a given .NET runtime major version, so recompilation is highly recommended to upgrade the app to run on a new .NET runtime major version. Then retest the app after upgrading to identify any issues.

Suppose upgrading via app recompilation isn't feasible. In that case, the .NET runtime provides [additional settings](#) to enable an app to run on a higher major .NET runtime version than the version it was compiled for. These settings don't change the risks involved in upgrading the app to a higher major .NET runtime version, and it's still required to retest the app post upgrade.

The .NET runtime supports loading libraries that target older .NET runtime versions. An app that's upgraded to a newer major .NET runtime version can reference libraries and NuGet packages that target older .NET runtime versions. It's unnecessary to simultaneously upgrade the target runtime version of all libraries and NuGet packages referenced by the app.

See also

- [Breaking changes in .NET](#)
- [Target frameworks](#)
- [.NET distribution packaging](#)
- [.NET Support Lifecycle Fact Sheet ↗](#)
- [Docker images for .NET ↗](#)

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

Select the .NET version to use

Article • 02/28/2023

This article explains the policies used by the .NET tools, SDK, and runtime for selecting versions. These policies provide a balance between running applications using the specified versions and enabling ease of upgrading both developer and end-user machines. These policies enable:

- Easy and efficient deployment of .NET, including security and reliability updates.
- Use the latest tools and commands independent of target runtime.

Version selection occurs:

- When you run an SDK command, [the SDK uses the latest installed version](#).
- When you build an assembly, [target framework monikers define build time APIs](#).
- When you run a .NET application, [target framework dependent apps roll-forward](#).
- When you publish a self-contained application, [self-contained deployments include the selected runtime](#).

The rest of this document examines those four scenarios.

The SDK uses the latest installed version

SDK commands include `dotnet new` and `dotnet run`. The .NET CLI must choose an SDK version for every `dotnet` command. It uses the latest SDK installed on the machine by default, even if:

- The project targets an earlier version of the .NET runtime.
- The latest version of the .NET SDK is a preview version.

You can take advantage of the latest SDK features and improvements while targeting earlier .NET runtime versions. You can target different runtime versions of .NET using the same SDK tools.

On rare occasions, you may need to use an earlier version of the SDK. You specify that version in a [*global.json* file](#). The "use latest" policy means you only use *global.json* to specify a .NET SDK version earlier than the latest installed version.

global.json can be placed anywhere in the file hierarchy. The CLI searches upward from the project directory for the first *global.json* it finds. You control which projects a given *global.json* applies to by its place in the file system. The .NET CLI searches for a *global.json* file iteratively navigating the path upward from the current working directory.

The first `global.json` file found specifies the version used. If that SDK version is installed, that version is used. If the SDK specified in the `global.json` isn't found, the .NET CLI uses [matching rules](#) to select a compatible SDK, or fails if none is found.

The following example shows the `global.json` syntax:

JSON

```
{  
  "sdk": {  
    "version": "5.0.0"  
  }  
}
```

The process for selecting an SDK version is:

1. `dotnet` searches for a `global.json` file iteratively reverse-navigating the path upward from the current working directory.
2. `dotnet` uses the SDK specified in the first `global.json` found.
3. `dotnet` uses the latest installed SDK if no `global.json` is found.

For more information about SDK version selection, see the [Matching rules](#) and [rollForward](#) sections of the [global.json overview](#) article.

Target Framework Monikers define build time APIs

You build your project against APIs defined in a **Target Framework Moniker** (TFM). You specify the [target framework](#) in the project file. Set the `TargetFramework` element in your project file as shown in the following example:

XML

```
<TargetFramework>net5.0</TargetFramework>
```

You may build your project against multiple TFMs. Setting multiple target frameworks is more common for libraries but can be done with applications as well. You specify a `TargetFrameworks` property (plural of `TargetFramework`). The target frameworks are semicolon-delimited as shown in the following example:

XML

```
<TargetFrameworks>net5.0;netcoreapp3.1;net47</TargetFrameworks>
```

A given SDK supports a fixed set of frameworks, capped to the target framework of the runtime it ships with. For example, the .NET 5 SDK includes the .NET 5 runtime, which is an implementation of the `net5.0` target framework. The .NET 5 SDK supports `netcoreapp2.0`, `netcoreapp2.1`, `netcoreapp3.0`, and so on, but not `net6.0` (or higher). You install the .NET 6 SDK to build for `net6.0`.

.NET Standard

.NET Standard was a way to target an API surface shared by different implementations of .NET. Starting with the release of .NET 5, which is an API standard itself, .NET Standard has little relevance, except for one scenario: .NET Standard is useful when you want to target both .NET and .NET Framework. .NET 5 implements all .NET Standard versions.

For more information, see [.NET 5 and .NET Standard](#).

Framework-dependent apps roll-forward

When you run an application from source with `dotnet run`, from a [framework-dependent deployment](#) with `dotnet myapp.dll`, or from a [framework-dependent executable](#) with `myapp.exe`, the `dotnet` executable is the **host** for the application.

The host chooses the latest patch version installed on the machine. For example, if you specified `net5.0` in your project file, and `5.0.2` is the latest .NET runtime installed, the `5.0.2` runtime is used.

If no acceptable `5.0.*` version is found, a new `5.*` version is used. For example, if you specified `net5.0` and only `5.1.0` is installed, the application runs using the `5.1.0` runtime. This behavior is referred to as "minor version roll-forward." Lower versions also won't be considered. When no acceptable runtime is installed, the application won't run.

A few usage examples demonstrate the behavior, if you target 5.0:

- ✅ 5.0 is specified. 5.0.3 is the highest patch version installed. 5.0.3 is used.
- ❌ 5.0 is specified. No 5.0.* versions are installed. 3.1.1 is the highest runtime installed. An error message is displayed.
- ✅ 5.0 is specified. No 5.0.* versions are installed. 5.1.0 is the highest runtime version installed. 5.1.0 is used.
- ❌ 3.0 is specified. No 3.x versions are installed. 5.0.0 is the highest runtime installed. An error message is displayed.

Minor version roll-forward has one side-effect that may affect end users. Consider the following scenario:

1. The application specifies that 5.0 is required.
2. When run, version 5.0.* isn't installed, however, 5.1.0 is. Version 5.1.0 will be used.
3. Later, the user installs 5.0.3 and runs the application again, 5.0.3 will now be used.

It's possible that 5.0.3 and 5.1.0 behave differently, particularly for scenarios like serializing binary data.

Control roll-forward behavior

Before overriding default roll-forward behavior, familiarize yourself with the level of [.NET runtime compatibility](#).

The roll-forward behavior for an application can be configured in four different ways:

1. Project-level setting by setting the `<RollForward>` property:

XML

```
<PropertyGroup>
  <RollForward>LatestMinor</RollForward>
</PropertyGroup>
```

2. The `*.runtimeconfig.json` file.

This file is produced when you compile your application. If the `<RollForward>` property was set in the project, it's reproduced in the `*.runtimeconfig.json` file as the `rollForward` setting. Users can edit this file to change the behavior of your application.

JSON

```
{
  "runtimeOptions": {
    "tfm": "net5.0",
    "rollForward": "LatestMinor",
    "framework": {
      "name": "Microsoft.NETCore.App",
      "version": "5.0.0"
    }
  }
}
```

3. The `dotnet` command's `--roll-forward <value>` property.

When you run an application, you can control the roll-forward behavior through the command line:

NET CLI

```
dotnet run --roll-forward LatestMinor  
dotnet myapp.dll --roll-forward LatestMinor  
myapp.exe --roll-forward LatestMinor
```

4. The `DOTNET_ROLL_FORWARD` environment variable.

Precedence

Roll forward behavior is set by the following order when your app is run, higher numbered items taking precedence over lower numbered items:

1. First the `*.runtimeconfig.json` config file is evaluated.
2. Next, the `DOTNET_ROLL_FORWARD` environment variable is considered, overriding the previous check.
3. Finally, any `--roll-forward` parameter passed to the running application overrides everything else.

Values

However you set the roll-forward setting, use one of the following values to set the behavior:

| Value | Description |
|-------------|--|
| Minor | Default if not specified. Roll-forward to the lowest higher minor version, if requested minor version is missing. If the requested minor version is present, then the <code>LatestPatch</code> policy is used. |
| Major | Roll-forward to the next available higher major version, and lowest minor version, if requested major version is missing. If the requested major version is present, then the <code>Minor</code> policy is used. |
| LatestPatch | Roll-forward to the highest patch version. This value disables minor version roll-forward. |
| LatestMinor | Roll-forward to highest minor version, even if requested minor version is present. |

| Value | Description |
|-------------|--|
| LatestMajor | Roll-forward to highest major and highest minor version, even if requested major is present. |
| Disable | Don't roll-forward, only bind to the specified version. This policy isn't recommended for general use since it disables the ability to roll-forward to the latest patches. This value is only recommended for testing. |

Self-contained deployments include the selected runtime

You can publish an application as a [self-contained distribution](#). This approach bundles the .NET runtime and libraries with your application. Self-contained deployments don't have a dependency on runtime environments. Runtime version selection occurs at publishing time, not run time.

The *restore* event that occurs when publishing selects the latest patch version of the given runtime family. For example, `dotnet publish` will select .NET 5.0.3 if it's the latest patch version in the .NET 5 runtime family. The target framework (including the latest installed security patches) is packaged with the application.

An error occurs if the minimum version specified for an application isn't satisfied. `dotnet publish` binds to the latest runtime patch version (within a given major.minor version family). `dotnet publish` doesn't support the roll-forward semantics of `dotnet run`. For more information about patches and self-contained deployments, see the article on [runtime patch selection](#) in deploying .NET applications.

Self-contained deployments may require a specific patch version. You can override the minimum runtime patch version (to higher or lower versions) in the project file, as shown in the following example:

XML

```
<PropertyGroup>
  <RuntimeFrameworkVersion>5.0.7</RuntimeFrameworkVersion>
</PropertyGroup>
```

The `RuntimeFrameworkVersion` element overrides the default version policy. For self-contained deployments, the `RuntimeFrameworkVersion` specifies the *exact* runtime framework version. For framework-dependent applications, the `RuntimeFrameworkVersion` specifies the *minimum* required runtime framework version.

See also

- [Download and install .NET.](#)
- [How to remove the .NET Runtime and SDK.](#)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

.NET Runtime configuration settings

Article • 11/11/2023

.NET 5+ (including .NET Core versions) supports the use of configuration files and environment variables to configure the behavior of .NET applications.

ⓘ Note

The articles in this section concern configuration of the .NET Runtime itself. If you're migrating to .NET Core 3.1 or later and are looking for a replacement for the *app.config* file, or if you simply want a way to use custom configuration values in your .NET app, see the [Microsoft.Extensions.Configuration.ConfigurationBuilder](#) class and [Configuration in .NET](#).

Using these settings is an attractive option if:

- You don't own or control the source code for an application and therefore are unable to configure it programmatically.
- Multiple instances of your application run at the same time on a single system, and you want to configure each for optimum performance.

.NET provides the following mechanisms for configuring behavior of the .NET runtime:

- The [runtimeconfig.json file](#)
- [MSBuild properties](#)
- [Environment variables](#)

ⓘ Tip

Configuring an option by using an environment variable applies the setting to all .NET apps. Configuring an option in the *runtimeconfig.json* or project file applies the setting to that application only.

Some configuration values can also be set programmatically by calling the [AppContext.SetSwitch](#) method.

The articles in this section of the documentation are organized by category, for example, [debugging](#) and [garbage collection](#). Where applicable, configuration options are shown for *runtimeconfig.json* files, MSBuild properties, environment variables, and, for cross-reference, *app.config* files for .NET Framework projects.

runtimeconfig.json

When a project is [built](#), an `[appname].runtimeconfig.json` file is generated in the output directory. If a `runtimeconfig.template.json` file exists in the same folder as the project file, any configuration options it contains are inserted into the `[appname].runtimeconfig.json` file. If you're building the app yourself, put any configuration options in the `runtimeconfig.template.json` file. If you're just running the app, insert them directly into the `[appname].runtimeconfig.json` file.

ⓘ Note

- The `[appname].runtimeconfig.json` file will get overwritten on subsequent builds.
- If your app's `OutputType` is not `Exe` and you want configuration options to be copied from `runtimeconfig.template.json` to `[appname].runtimeconfig.json`, you must explicitly set `GenerateRuntimeConfigurationFiles` to `true` in your project file. For apps that require a `runtimeconfig.json` file, this property defaults to `true`.

Specify runtime configuration options in the `configProperties` section of the `runtimeconfig.json` or `runtimeconfig.template.json` file. This section has the form:

JSON

```
"configProperties": {  
    "config-property-name1": "config-value1",  
    "config-property-name2": "config-value2"  
}
```

Example `[appname].runtimeconfig.json` file

If you're placing the options in the output JSON file, nest them under the `runtimeOptions` property.

JSON

```
{  
    "runtimeOptions": {  
        "tfm": "netcoreapp3.1",  
        "framework": {  
            "name": "Microsoft.NETCore.App",  
            "version": "3.1.0"  
        }  
    }  
}
```

```
        },
        "configProperties": {
            "System.Globalization.UseNls": true,
            "System.Net.DisableIPv6": true,
            "System.GC.Concurrent": false,
            "System.Threading.ThreadPool.MinThreads": 4,
            "System.Threading.ThreadPool.MaxThreads": 25
        }
    }
}
```

Example `runtimeconfig.template.json` file

If you're placing the options in the template JSON file, omit the `runtimeOptions` property.

JSON

```
{
    "configProperties": {
        "System.Globalization.UseNls": true,
        "System.Net.DisableIPv6": true,
        "System.GC.Concurrent": false,
        "System.Threading.ThreadPool.MinThreads": "4",
        "System.Threading.ThreadPool.MaxThreads": "25"
    }
}
```

MSBuild properties

Some runtime configuration options can be set using MSBuild properties in the `.csproj` or `.vbproj` file of SDK-style .NET projects. MSBuild properties take precedence over options set in the `runtimeconfig.template.json` file.

For runtime configuration settings that don't have a specific MSBuild property, you can use the `RuntimeHostConfigurationOption` MSBuild item instead. Use the `runtimeconfig.json` setting name as the value of the `Include` attribute.

Here is an example SDK-style project file with MSBuild properties for configuring the behavior of the .NET runtime:

XML

```
<Project Sdk="Microsoft.NET.Sdk">

    <PropertyGroup>
```

```
<OutputType>Exe</OutputType>
<TargetFramework>netcoreapp3.1</TargetFramework>
</PropertyGroup>

<PropertyGroup>
  <ConcurrentGarbageCollection>false</ConcurrentGarbageCollection>
  <ThreadPoolMinThreads>4</ThreadPoolMinThreads>
  <ThreadPoolMaxThreads>25</ThreadPoolMaxThreads>
</PropertyGroup>

<ItemGroup>
  <RuntimeHostConfigurationOption Include="System.Globalization.UseNls"
Value="true" />
  <RuntimeHostConfigurationOption Include="System.Net.DisableIPv6"
Value="true" />
</ItemGroup>

</Project>
```

MSBuild properties for configuring the behavior of the runtime are noted in the individual articles for each area, for example, [garbage collection](#). They're also listed in the [Runtime configuration](#) section of the MSBuild properties reference for SDK-style projects.

Environment variables

Environment variables can be used to supply some runtime configuration information. Configuration knobs specified as environment variables generally have the prefix `DOTNET_`.

Note

.NET 6 standardizes on the prefix `DOTNET_` instead of `COMPlus_` for environment variables that configure .NET run-time behavior. However, the `COMPlus_` prefix will continue to work. If you're using a previous version of the .NET runtime, you should still use the `COMPlus_` prefix for environment variables.

You can define environment variables from the Windows Control Panel, at the command line, or programmatically by calling the [`Environment.SetEnvironmentVariable\(String, String\)`](#) method on both Windows and Unix-based systems.

The following examples show how to set an environment variable at the command line:

shell

```
# Windows  
set DOTNET_GCRetainVM=1  
  
# Powershell  
$env:DOTNET_GCRetainVM="1"  
  
# Unix  
export DOTNET_GCRetainVM=1
```

See also

- [.NET environment variables](#)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

Runtime configuration options for compilation

Article • 11/11/2023

This article details the settings you can use to configure .NET compilation.

ⓘ Note

.NET 6 standardizes on the prefix `DOTNET_` instead of `COMPlus_` for environment variables that configure .NET run-time behavior. However, the `COMPlus_` prefix will continue to work. If you're using a previous version of the .NET runtime, you should still use the `COMPlus_` prefix for environment variables.

Tiered compilation

- Configures whether the just-in-time (JIT) compiler uses [tiered compilation](#). Tiered compilation transitions methods through two tiers:
 - The first tier generates code more quickly ([quick JIT](#)) or loads pre-compiled code ([ReadyToRun](#)).
 - The second tier generates optimized code in the background ("optimizing JIT").
- In .NET Core 3.0 and later, tiered compilation is enabled by default.
- In .NET Core 2.1 and 2.2, tiered compilation is disabled by default.
- For more information, see the [Tiered compilation guide ↗](#).

| Setting name | Values |
|-----------------------------|---|
| runtimconfig.json | <code>System.Runtime.TieredCompilation</code> <code>true</code> - enabled <code>false</code> - disabled |
| MSBuild property | <code>TieredCompilation</code> <code>true</code> - enabled <code>false</code> - disabled |
| Environment variable | <code>COMPlus_TieredCompilation</code> or <code>DOTNET_TieredCompilation</code> <code>1</code> - enabled <code>0</code> - disabled |

Examples

runtimconfig.json file:

JSON

```
{  
    "runtimeOptions": {  
        "configProperties": {  
            "System.Runtime.TieredCompilation": false  
        }  
    }  
}
```

runtimeconfig.template.json file:

JSON

```
{  
    "configProperties": {  
        "System.Runtime.TieredCompilation": false  
    }  
}
```

Project file:

XML

```
<Project Sdk="Microsoft.NET.Sdk">  
  
    <PropertyGroup>  
        <TieredCompilation>false</TieredCompilation>  
    </PropertyGroup>  
  
</Project>
```

Quick JIT

- Configures whether the JIT compiler uses *quick JIT*. For methods that don't contain loops and for which pre-compiled code is not available, quick JIT compiles them more quickly but without optimizations.
- Enabling quick JIT decreases startup time but can produce code with degraded performance characteristics. For example, the code may use more stack space, allocate more memory, and run slower.
- If quick JIT is disabled but [tiered compilation](#) is enabled, only pre-compiled code participates in tiered compilation. If a method is not pre-compiled with [ReadyToRun](#), the JIT behavior is the same as if [tiered compilation](#) were disabled.
- In .NET Core 3.0 and later, quick JIT is enabled by default.
- In .NET Core 2.1 and 2.2, quick JIT is disabled by default.

| Setting name | | Values |
|-----------------------------|---|------------------------------------|
| runtimeconfig.json | System.Runtime.TieredCompilation.QuickJit | true - enabled false - disabled |
| MSBuild property | TieredCompilationQuickJit | true - enabled false - disabled |
| Environment variable | COMPlus_TC_QuickJit or DOTNET_TC_QuickJit | 1 - enabled 0 - disabled |

Examples

runtimeconfig.json file:

JSON

```
{
  "runtimeOptions": {
    "configProperties": {
      "System.Runtime.TieredCompilation.QuickJit": false
    }
  }
}
```

runtimeconfig.template.json file:

JSON

```
{
  "configProperties": {
    "System.Runtime.TieredCompilation.QuickJit": false
  }
}
```

Project file:

XML

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <TieredCompilationQuickJit>false</TieredCompilationQuickJit>
  </PropertyGroup>

</Project>
```

Quick JIT for loops

- Configures whether the JIT compiler uses quick JIT on methods that contain loops.
- Enabling quick JIT for loops may improve startup performance. However, long-running loops can get stuck in less-optimized code for long periods.
- If [quick JIT](#) is disabled, this setting has no effect.
- If you omit this setting, quick JIT is not used for methods that contain loops. This is equivalent to setting the value to `false`.

| | Setting name | Values |
|-----------------------------|--|--|
| runtimconfig.json | <code>System.Runtime.TieredCompilation.QuickJitForLoops</code> | <code>false</code> - disabled <code>true</code> - enabled |
| MSBuild property | <code>TieredCompilationQuickJitForLoops</code> | <code>false</code> - disabled <code>true</code> - enabled |
| Environment variable | <code>COMPlus_TC_QuickJitForLoops</code> or <code>DOTNET_TC_QuickJitForLoops</code> | <code>0</code> - disabled <code>1</code> - enabled |

Examples

runtimconfig.json file:

```
JSON

{
  "runtimeOptions": {
    "configProperties": {
      "System.Runtime.TieredCompilation.QuickJitForLoops": false
    }
  }
}
```

runtimconfig.template.json file:

```
JSON

{
  "configProperties": {
    "System.Runtime.TieredCompilation.QuickJitForLoops": false
  }
}
```

```
}
```

Project file:

XML

```
<Project Sdk="Microsoft.NET.Sdk">

<PropertyGroup>

<TieredCompilationQuickJitForLoops>true</TieredCompilationQuickJitForLoops>
</PropertyGroup>

</Project>
```

ReadyToRun

- Configures whether the .NET Core runtime uses pre-compiled code for images with available ReadyToRun data. Disabling this option forces the runtime to JIT-compile framework code.
- For more information, see [Ready to Run](#).
- If you omit this setting, .NET uses ReadyToRun data when it's available. This is equivalent to setting the value to `1`.

| | Setting name | Values |
|-----------------------------|---|---|
| Environment variable | <code>COMPlus_ReadyToRun</code> or <code>DOTNET_ReadyToRun</code> | <code>1</code> - enabled <code>0</code> - disabled |

Profile-guided optimization

This setting enables dynamic or tiered profile-guided optimization (PGO) in .NET 6 and later versions.

| | Setting name | Values |
|-----------------------------|-------------------------------|--|
| Environment variable | <code>DOTNET_TieredPGO</code> | <code>1</code> - enabled <code>0</code> - disabled |
| MSBuild property | <code>TieredPGO</code> | <code>true</code> - enabled <code>false</code> - disabled |

Examples

Project file:

```
XML

<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <TieredPGO>true</TieredPGO>
  </PropertyGroup>

</Project>
```

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

Runtime configuration options for debugging and profiling

Article • 12/05/2023

This article details the settings you can use to configure .NET debugging and profiling.

ⓘ Note

.NET 6 standardizes on the prefix `DOTNET_` instead of `COMPlus_` for environment variables that configure .NET run-time behavior. However, the `COMPlus_` prefix will continue to work. If you're using a previous version of the .NET runtime, you should still use the `COMPlus_` prefix for environment variables.

Enable diagnostics

- Configures whether the debugger, the profiler, and EventPipe diagnostics are enabled or disabled.
- If you omit this setting, diagnostics are enabled. This is equivalent to setting the value to `1`.

[+] Expand table

| Setting name | Values |
|--------------------------------|--|
| <code>runtimconfig.json</code> | N/A |
| Environment variable | <code>COMPlus_EnableDiagnostics</code> or <code>DOTNET_EnableDiagnostics</code> <code>1</code> - enabled <code>0</code> - disabled |

Enable profiling

- Configures whether profiling is enabled for the currently running process.
- If you omit this setting, profiling is disabled. This is equivalent to setting the value to `0`.

[+] Expand table

| | Setting name | Values |
|-----------------------------|--------------------------|--|
| runtimeconfig.json | N/A | N/A |
| Environment variable | CORECLR_ENABLE_PROFILING | <input type="radio"/> 0 - disabled <input checked="" type="radio"/> 1 - enabled |

Profiler GUID

- Specifies the GUID of the profiler to load into the currently running process.

[Expand table](#)

| | Setting name | Values |
|-----------------------------|---------------------|--------------------|
| runtimeconfig.json | N/A | N/A |
| Environment variable | CORECLR_PROFILER | <i>string-guid</i> |

Profiler location

- Specifies the path to the profiler DLL to load into the currently running process (or 32-bit or 64-bit process).
- If more than one variable is set, the bitness-specific variables take precedence. They specify which bitness of profiler to load.
- For more information, see [Finding the profiler library](#).

[Expand table](#)

| | Setting name | Values |
|-----------------------------|--------------------------|--------------------|
| Environment variable | CORECLR_PROFILER_PATH | <i>string-path</i> |
| Environment variable | CORECLR_PROFILER_PATH_32 | <i>string-path</i> |
| Environment variable | CORECLR_PROFILER_PATH_64 | <i>string-path</i> |

Export perf maps and jit dumps

- Enables or disables selective enablement of perf maps or jit dumps. These files allow third party tools, such as the Linux `perf` tool, to identify call sites for dynamically generated code and precompiled ReadyToRun (R2R) modules.

- If you omit this setting, writing perf map and jit dump files are both disabled. This is equivalent to setting the value to `0`.
- When perf maps are disabled, not all managed callsites will be properly resolved.
- Depending on the Linux kernel version, both formats are supported by the `perf` tool.
- Enabling perf maps or jit dumps causes a 10-20% overhead. To minimize performance impact, it's recommended to selectively enable either perf maps or jit dumps, but not both.

The following table compares perf maps and jit maps.

[\[+\] Expand table](#)

| Format | Description | Supported on |
|------------------|---|---|
| <i>Perf maps</i> | Emits <code>/tmp/perf-<pid>.map</code> , which contains symbolic information for dynamically generated code. Emits <code>/tmp/perfinfo-<pid>.map</code> , which includes ReadyToRun (R2R) module symbol information and is used by PerfCollect . | Perf maps are supported on all Linux kernel versions. |
| <i>Jit dumps</i> | The jit dump format supersedes perf maps and contains more detailed symbolic information. When enabled, jit dumps are output to <code>/tmp/jit-<pid>.dump</code> files. | Linux kernel versions 5.4 or higher. |

[\[+\] Expand table](#)

| | Setting name | Values |
|---------------------------------|--|--|
| <code>runtimeconfig.json</code> | N/A | N/A |
| Environment variable | <code>COMPlus_PerfMapEnabled</code> or <code>DOTNET_PerfMapEnabled</code> | <code>0</code> - disabled <code>1</code> - perf maps and jit dumps both enabled <code>2</code> - jit dumps enabled <code>3</code> - perf maps enabled |

Perf log markers

- Enables or disables the specified signal to be accepted and ignored as a marker in the perf logs.
- If you omit this setting, the specified signal is not ignored. This is equivalent to setting the value to `0`.

| Setting name | Values |
|-----------------------------|---|
| runtimeconfig.json | N/A |
| Environment variable | <code>COMPlus_PerfMapIgnoreSignal</code> or <code>DOTNET_PerfMapIgnoreSignal</code> 0 - disabled 1 - enabled |

Note

This setting is ignored if `DOTNET_PerfMapEnabled` is omitted or set to 0 (that is, disabled).

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



.NET feedback

.NET is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

Runtime configuration options for garbage collection

Article • 11/11/2023

This page contains information about settings for the .NET runtime garbage collector (GC). If you're trying to achieve peak performance of a running app, consider using these settings. However, the defaults provide optimum performance for most applications in typical situations.

Settings are arranged into groups on this page. The settings within each group are commonly used in conjunction with each other to achieve a specific result.

Note

- These configurations are only read by the runtime when the GC is initialized (usually this means during the process startup time). If you change an environment variable when a process is already running, the change won't be reflected in that process. Settings that can be changed through APIs at run time, such as `latency level`, are omitted from this page.
- Because GC is per process, it rarely ever makes sense to set these configurations at the machine level. For example, you wouldn't want every .NET process on a machine to use server GC or the same heap hard limit.
- For number values, use decimal notation for settings in the `runtimeconfig.json` or `runtimeconfig.template.json` file and hexadecimal notation for environment variable settings. For hexadecimal values, you can specify them with or without the "0x" prefix.
- If you're using the environment variables, .NET 6 and later versions standardize on the prefix `DOTNET_` instead of `COMPlus_`. However, the `COMPlus_` prefix will continue to work. If you're using a previous version of the .NET runtime, you should still use the `COMPlus_` prefix, for example, `COMPlus_gcServer`.

Ways to specify the configuration

For different versions of the .NET runtime, there are different ways to specify the configuration values. The following table shows a summary.

| Config location | .NET versions this location applies to | Formats | How it's interpreted |
|---|--|----------|--|
| runtimeconfig.json file/ runtimeconfig.template.json file | .NET (Core) | n | n is interpreted as a decimal value. |
| Environment variable | .NET Framework, .NET (Core) | 0xn or n | n is interpreted as a hex value in either format |
| app.config file | .NET Framework | 0xn | n is interpreted as a hex value ¹ |

¹ You can specify a value without the `0x` prefix for an app.config file setting, but it's not recommended. On .NET Framework 4.8+, due to a bug, a value specified without the `0x` prefix is interpreted as hexadecimal, but on previous versions of .NET Framework, it's interpreted as decimal. To avoid having to change your config, use the `0x` prefix when specifying a value in your app.config file.

For example, to specify 12 heaps for `GCHeapCount` for a .NET Framework app named `A.exe`, add the following XML to the `A.exe.config` file.

XML

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
    ...
    <runtime>
        <gcServer enabled="true"/>
        <GCHeapCount>0xc</GCHeapCount>
    </runtime>
</configuration>
```

For both .NET (Core) and .NET Framework, you can use environment variables.

On Windows using .NET 6 or a later version:

Windows Command Prompt

```
SET DOTNET_gcServer=1
SET DOTNET_GCHeapCount=c
```

On Windows using .NET 5 or earlier:

Windows Command Prompt

```
SET COMPlus_gcServer=1
SET COMPlus_GCHeapCount=c
```

On other operating systems:

For .NET 6 or later versions:

Bash

```
export DOTNET_gcServer=1
export DOTNET_GCHeapCount=c
```

For .NET 5 and earlier versions:

Bash

```
export COMPlus_gcServer=1
export COMPlus_GCHeapCount=c
```

If you're not using .NET Framework, you can also set the value in the *runtimeconfig.json* or *runtimeconfig.template.json* file.

runtimeconfig.json file:

JSON

```
{
  "runtimeOptions": {
    "configProperties": {
      "System.GC.Server": true,
      "System.GC.HeapCount": 12
    }
  }
}
```

runtimeconfig.template.json file:

JSON

```
{
  "configProperties": {
    "System.GC.Server": true,
    "System.GC.HeapCount": 12
  }
}
```

Flavors of garbage collection

The two main flavors of garbage collection are workstation GC and server GC. For more information about differences between the two, see [Workstation and server garbage collection](#).

The subflavors of garbage collection are background and non-concurrent.

Use the following settings to select flavors of garbage collection:

- [Workstation vs. server GC](#)
- [Background GC](#)

Workstation vs. server

- Configures whether the application uses workstation garbage collection or server garbage collection.
- Default: Workstation garbage collection. This is equivalent to setting the value to `false`.

[+] [Expand table](#)

| | Setting name | Values | Version introduced |
|--------------------------------------|--------------------------------------|--|---------------------------|
| runtimeconfig.json | <code>System.GC.Server</code> | <code>false</code> - workstation <code>true</code> - server | .NET Core 1.0 |
| MSBuild property | <code>ServerGarbageCollection</code> | <code>false</code> - workstation <code>true</code> - server | .NET Core 1.0 |
| Environment variable | <code>COMPlus_gcServer</code> | <code>0</code> - workstation <code>1</code> - server | .NET Core 1.0 |
| Environment variable | <code>DOTNET_gcServer</code> | <code>0</code> - workstation <code>1</code> - server | .NET 6 |
| app.config for .NET Framework | <code>GCServer</code> | <code>false</code> - workstation <code>true</code> - server | |

Examples

runtimeconfig.json file:

JSON

```
{  
    "runtimeOptions": {  
        "configProperties": {  
            "System.GC.Server": true  
        }  
    }  
}
```

runtimeconfig.template.json file:

JSON

```
{  
    "configProperties": {  
        "System.GC.Server": true  
    }  
}
```

Project file:

XML

```
<Project Sdk="Microsoft.NET.Sdk">  
  
    <PropertyGroup>  
        <ServerGarbageCollection>true</ServerGarbageCollection>  
    </PropertyGroup>  
  
</Project>
```

Background GC

- Configures whether background (concurrent) garbage collection is enabled.
- Default: Use background GC. This is equivalent to setting the value to `true`.
- For more information, see [Background garbage collection](#).

 Expand table

| Setting name | Values | Version introduced |
|--------------------------------------|--|--------------------|
| runtimesconfig.json | System.GC.Concurrent true - background GC false - non-concurrent GC | .NET Core 1.0 |
| MSBuild property | ConcurrentGarbageCollection true - background GC false - non-concurrent GC | .NET Core 1.0 |
| Environment variable | COMPlus_gcConcurrent 1 - background GC 0 - non-concurrent GC | .NET Core 1.0 |
| Environment variable | DOTNET_gcConcurrent 1 - background GC 0 - non-concurrent GC | .NET 6 |
| app.config for .NET Framework | gcConcurrent true - background GC false - non-concurrent GC | |

Examples

runtimesconfig.json file:

JSON

```
{
  "runtimeOptions": {
    "configProperties": {
      "System.GC.Concurrent": false
    }
  }
}
```

runtimesconfig.template.json file:

JSON

```
{
  "configProperties": {
```

```
        "System.GC.Concurrent": false  
    }  
}
```

Project file:

XML

```
<Project Sdk="Microsoft.NET.Sdk">  
  
    <PropertyGroup>  
        <ConcurrentGarbageCollection>false</ConcurrentGarbageCollection>  
    </PropertyGroup>  
  
</Project>
```

Manage resource usage

Use the following settings to manage the garbage collector's memory and processor usage:

- [Affinitize](#)
- [Affinimize mask](#)
- [Affinimize ranges](#)
- [CPU groups](#)
- [Heap count](#)
- [Heap limit](#)
- [Heap limit percent](#)
- [High memory percent](#)
- [Per-object-heap limits](#)
- [Per-object-heap limit percents](#)
- [Retain VM](#)

For more information about some of these settings, see the [Middle ground between workstation and server GC](#) blog entry.

Heap count

- Limits the number of heaps created by the garbage collector.
- Applies to server garbage collection only.
- If [GC processor affinity](#) is enabled, which is the default, the heap count setting affinizes n GC heaps/threads to the first n processors. (Use the [affinimize mask](#) or [affinimize ranges](#) settings to specify exactly which processors to affinimize.)

- If [GC processor affinity](#) is disabled, this setting limits the number of GC heaps.
- For more information, see the [GCHeapCount remarks](#).

[\[+\] Expand table](#)

| | Setting name | Values | Version introduced |
|--------------------------------------|----------------------------------|--------------------------|----------------------|
| runtimesconfig.json | <code>System.GC.HeapCount</code> | <i>decimal value</i> | .NET Core 3.0 |
| Environment variable | <code>COMPlus_GCHeapCount</code> | <i>hexadecimal value</i> | .NET Core 3.0 |
| Environment variable | <code>DOTNET_GCHeapCount</code> | <i>hexadecimal value</i> | .NET 6 |
| app.config for .NET Framework | <code>GCHeapCount</code> | <i>decimal value</i> | .NET Framework 4.6.2 |

This configuration setting doesn't have a specific MSBuild property. However, you can add a `RuntimeHostConfigurationOption` MSBuild item instead. Use the *runtimesconfig.json* setting name as the value of the `Include` attribute. For an example, see [MSBuild properties](#).

Examples

runtimesconfig.json file:

```
JSON
{
  "runtimeOptions": {
    "configProperties": {
      "System.GC.HeapCount": 16
    }
  }
}
```

runtimesconfig.template.json file:

```
JSON
{
  "configProperties": {
    "System.GC.HeapCount": 16
  }
}
```

Tip

If you're setting the option in `runtimeconfig.json`, specify a decimal value. If you're setting the option as an environment variable, specify a hexadecimal value. For example, to limit the number of heaps to 16, the values would be 16 for the JSON file and 0x10 or 10 for the environment variable.

Affinize mask

- Specifies the exact processors that garbage collector threads should use.
- If [GC processor affinity](#) is disabled, this setting is ignored.
- Applies to server garbage collection only.
- The value is a bit mask that defines the processors that are available to the process. For example, a decimal value of 1023 (or a hexadecimal value of 0x3FF or 3FF if you're using the environment variable) is 0011 1111 1111 in binary notation. This specifies that the first 10 processors are to be used. To specify the next 10 processors, that is, processors 10-19, specify a decimal value of 1047552 (or a hexadecimal value of 0xFFC00 or FFC00), which is equivalent to a binary value of 1111 1111 1100 0000 0000.

[\[+\] Expand table](#)

| | Setting name | Values | Version introduced |
|--------------------------------------|--|--------------------------|----------------------|
| runtimeconfig.json | <code>System.GC.HeapAffinizeMask</code> | <i>decimal value</i> | .NET Core 3.0 |
| Environment variable | <code>COMPlus_GCHheapAffinizeMask</code> | <i>hexadecimal value</i> | .NET Core 3.0 |
| Environment variable | <code>DOTNET_GCHheapAffinizeMask</code> | <i>hexadecimal value</i> | .NET 6 |
| app.config for .NET Framework | <code>GCHeapAffinizeMask</code> | <i>decimal value</i> | .NET Framework 4.6.2 |

This configuration setting doesn't have a specific MSBuild property. However, you can add a `RuntimeHostConfigurationOption` MSBuild item instead. Use the `runtimeconfig.json` setting name as the value of the `Include` attribute. For an example, see [MSBuild properties](#).

Examples

runtimeconfig.json file:

```
JSON

{
  "runtimeOptions": {
    "configProperties": {
      "System.GC.HeapAffinitizeMask": 1023
    }
  }
}
```

runtimeconfig.template.json file:

```
JSON

{
  "configProperties": {
    "System.GC.HeapAffinitizeMask": 1023
  }
}
```

Affinitize ranges

- Specifies the list of processors to use for garbage collector threads.
- This setting is similar to [System.GC.HeapAffinitizeMask](#), except it allows you to specify more than 64 processors.
- For Windows operating systems, prefix the processor number or range with the corresponding [CPU group](#), for example, "0:1-10,0:12,1:50-52,1:7". If you don't actually have more than 1 CPU group, you can't use this setting. You must use the [Affinitize mask](#) setting. And the numbers you specify are within that group, which means it cannot be ≥ 64 .
- For Linux operating systems, where the [CPU group](#) concept doesn't exist, you can use both this setting and the [Affinitize mask](#) setting to specify the same ranges. And instead of "0:1-10", specify "1-10" because you don't need to specify a group index.
- If [GC processor affinity](#) is disabled, this setting is ignored.
- Applies to server garbage collection only.
- For more information, see [Making CPU configuration better for GC on machines with > 64 CPUs](#) on Maoni Stephens' blog.

[] [Expand table](#)

| Setting name | Values | Version introduced |
|-----------------------------|---|--------------------|
| runtimconfig.json | System.GC.HeapAffinizeRanges Comma-separated list of processor numbers or ranges of processor numbers. Unix example: "1-10,12,50-52,70" Windows example: "0:1-10,0:12,1:50-52,1:7" | .NET Core 3.0 |
| Environment variable | COMPlus_GCHepAffinizeRanges Comma-separated list of processor numbers or ranges of processor numbers. Unix example: "1-10,12,50-52,70" Windows example: "0:1-10,0:12,1:50-52,1:7" | .NET Core 3.0 |
| Environment variable | DOTNET_GCHepAffinizeRanges Comma-separated list of processor numbers or ranges of processor numbers. Unix example: "1-10,12,50-52,70" Windows example: "0:1-10,0:12,1:50-52,1:7" | .NET 6 |

This configuration setting doesn't have a specific MSBuild property. However, you can add a `RuntimeHostConfigurationOption` MSBuild item instead. Use the `runtimconfig.json` setting name as the value of the `Include` attribute. For an example, see [MSBuild properties](#).

Examples

`runtimconfig.json` file:

```
JSON
{
  "runtimeOptions": {
    "configProperties": {
      "System.GC.HeapAffinizeRanges": "0:1-10,0:12,1:50-52,1:70"
    }
  }
}
```

```
        }
    }
}
```

runtimeconfig.template.json file:

JSON

```
{
  "configProperties": {
    "System.GC.HeapAffinimizeRanges": "0:1-10,0:12,1:50-52,1:70"
  }
}
```

CPU groups

- Configures whether the garbage collector uses [CPU groups](#) or not.

When a 64-bit Windows computer has multiple CPU groups, that is, there are more than 64 processors, enabling this element extends garbage collection across all CPU groups. The garbage collector uses all cores to create and balance heaps.

Note

This is a Windows-only concept. In older Windows versions, Windows limited a process to one CPU group. Thus, GC only used one CPU group unless you used this setting to enable multiple CPU groups. This OS limitation was lifted in Windows 11 and Server 2022. Also, starting in .NET 7, GC by default uses all CPU groups when running on Windows 11 or Server 2022.

- Applies to server garbage collection on 64-bit Windows operating systems only.
- Default: GC does not extend across CPU groups. This is equivalent to setting the value to `0`.
- For more information, see [Making CPU configuration better for GC on machines with > 64 CPUs](#) on Maoni Stephens' blog.

 Expand table

| Setting name | Values | Version introduced |
|---------------------------|--|--------------------|
| runtimeconfig.json | System.GC.CpuGroup false - disabled | .NET 5 |

| | Setting name | Values | Version introduced |
|--------------------------------------|---------------------------------|--|---------------------------|
| | | <code>true</code> - enabled | |
| Environment variable | <code>COMPlus_GCCpuGroup</code> | <code>0</code> - disabled <code>1</code> - enabled | .NET Core 1.0 |
| Environment variable | <code>DOTNET_GCCpuGroup</code> | <code>0</code> - disabled <code>1</code> - enabled | .NET 6 |
| app.config for .NET Framework | <code>GCCpuGroup</code> | <code>false</code> - disabled <code>true</code> - enabled | |

This configuration setting doesn't have a specific MSBuild property. However, you can add a `RuntimeHostConfigurationOption` MSBuild item instead. Use the `runtimeconfig.json` setting name as the value of the `Include` attribute. For an example, see [MSBuild properties](#).

ⓘ Note

To configure the common language runtime (CLR) to also distribute threads from the thread pool across all CPU groups, enable the `Thread_UseAllCpuGroups` element option. For .NET Core apps, you can enable this option by setting the value of the `DOTNET_Thread_UseAllCpuGroups` environment variable to `1`.

Affinize

- Specifies whether to *affinize* garbage collection threads with processors. To affinize a GC thread means that it can only run on its specific CPU. A heap is created for each GC thread.
- Applies to server garbage collection only.
- Default: Affinize garbage collection threads with processors. This is equivalent to setting the value to `false`.

[] Expand table

| | Setting name | Values | Version introduced |
|---------------------------|-----------------------------------|---|---------------------------|
| runtimeconfig.json | <code>System.GC.NoAffinize</code> | <code>false</code> - affinize <code>true</code> - don't affinize | .NET Core 3.0 |

| Setting name | Values | Version introduced |
|--------------------------------------|---|----------------------|
| Environment variable | <code>COMPlus_GCNoAffinitize</code> <input type="radio"/> 0 - affinitize <input checked="" type="radio"/> 1 - don't affinitize | .NET Core 3.0 |
| Environment variable | <code>DOTNET_GCNoAffinitize</code> <input type="radio"/> 0 - affinitize <input checked="" type="radio"/> 1 - don't affinitize | .NET 6 |
| app.config for .NET Framework | <code>GCNoAffinitize</code> <input type="radio"/> <code>false</code> - affinitize <input checked="" type="radio"/> <code>true</code> - don't affinitize | .NET Framework 4.6.2 |

This configuration setting doesn't have a specific MSBuild property. However, you can add a `RuntimeHostConfigurationOption` MSBuild item instead. Use the `runtimeconfig.json` setting name as the value of the `Include` attribute. For an example, see [MSBuild properties](#).

Examples

`runtimeconfig.json` file:

JSON

```
{
  "runtimeOptions": {
    "configProperties": {
      "System.GC.NoAffinitize": true
    }
  }
}
```

`runtimeconfig.template.json` file:

JSON

```
{
  "configProperties": {
    "System.GC.NoAffinitize": true
  }
}
```

Heap limit

- Specifies the maximum commit size, in bytes, for the GC heap and GC bookkeeping.
- This setting only applies to 64-bit computers.
- This setting is ignored if the [Per-object-heap limits](#) are configured.
- The default value, which only applies in certain cases, is the greater of 20 MB or 75% of the memory limit on the container. The default value applies if:
 - The process is running inside a container that has a specified memory limit.
 - `System.GC.HeapHardLimitPercent` is not set.

[] Expand table

| | Setting name | Values | Version introduced |
|-----------------------------|--------------------------------------|--------------------------|---------------------------|
| runtimesconfig.json | <code>System.GC.HeapHardLimit</code> | <i>decimal value</i> | .NET Core 3.0 |
| Environment variable | <code>COMPlus_GCHardLimit</code> | <i>hexadecimal value</i> | .NET Core 3.0 |
| Environment variable | <code>DOTNET_GCHardLimit</code> | <i>hexadecimal value</i> | .NET 6 |

This configuration setting doesn't have a specific MSBuild property. However, you can add a `RuntimeHostConfigurationOption` MSBuild item instead. Use the `runtimesconfig.json` setting name as the value of the `Include` attribute. For an example, see [MSBuild properties](#).

Examples

`runtimesconfig.json` file:

JSON

```
{
  "runtimeOptions": {
    "configProperties": {
      "System.GC.HeapHardLimit": 209715200
    }
  }
}
```

`runtimesconfig.template.json` file:

JSON

```
{  
  "configProperties": {  
    "System.GC.HeapHardLimit": 209715200  
  }  
}
```

💡 Tip

If you're setting the option in `runtimeconfig.json`, specify a decimal value. If you're setting the option as an environment variable, specify a hexadecimal value. For example, to specify a heap hard limit of 200 mebibytes (MiB), the values would be 209715200 for the JSON file and 0xC800000 or C800000 for the environment variable.

Heap limit percent

- Specifies the allowable GC heap usage as a percentage of the total physical memory.
- If `System.GC.HeapHardLimit` is also set, this setting is ignored.
- This setting only applies to 64-bit computers.
- If the process is running inside a container that has a specified memory limit, the percentage is calculated as a percentage of that memory limit.
- This setting is ignored if the [Per-object-heap limits](#) are configured.
- The default value, which only applies in certain cases, is the greater of 20 MB or 75% of the memory limit on the container. The default value applies if:
 - The process is running inside a container that has a specified memory limit.
 - `System.GC.HeapHardLimit` is not set.

[+] Expand table

| Setting name | Values | Version introduced |
|-----------------------------------|--|--------------------|
| <code>runtimeconfig.json</code> | <code>System.GC.HeapHardLimitPercent</code> <i>decimal value</i> | .NET Core 3.0 |
| <code>Environment variable</code> | <code>COMPlus_GCHardLimitPercent</code> <i>hexadecimal value</i> | .NET Core 3.0 |

| Setting name | Values | Version introduced |
|-----------------------------|---|--------------------|
| Environment variable | <code>DOTNET_GCHeapHardLimitPercent</code> hexadecimal value | .NET 6 |

This configuration setting doesn't have a specific MSBuild property. However, you can add a `RuntimeHostConfigurationOption` MSBuild item instead. Use the `runtimeconfig.json` setting name as the value of the `Include` attribute. For an example, see [MSBuild properties](#).

Examples

`runtimeconfig.json` file:

JSON

```
{
  "runtimeOptions": {
    "configProperties": {
      "System.GC.HeapHardLimitPercent": 30
    }
  }
}
```

`runtimeconfig.template.json` file:

JSON

```
{
  "configProperties": {
    "System.GC.HeapHardLimitPercent": 30
  }
}
```

Tip

If you're setting the option in `runtimeconfig.json`, specify a decimal value. If you're setting the option as an environment variable, specify a hexadecimal value. For example, to limit the heap usage to 30%, the values would be 30 for the JSON file and 0x1E or 1E for the environment variable.

Per-object-heap limits

You can specify the GC's allowable heap usage on a per-object-heap basis. The different heaps are the large object heap (LOH), small object heap (SOH), and pinned object heap (POH).

- If you specify a value for any of the `DOTNET_GCHeapHardLimitSOH`, `DOTNET_GCHeapHardLimitLOH`, or `DOTNET_GCHeapHardLimitPOH` settings, you must also specify a value for `DOTNET_GCHeapHardLimitSOH` and `DOTNET_GCHeapHardLimitLOH`. If you don't, the runtime will fail to initialize.
- The default value for `DOTNET_GCHeapHardLimitPOH` is 0. `DOTNET_GCHeapHardLimitSOH` and `DOTNET_GCHeapHardLimitLOH` don't have default values.

[+] Expand table

| | Setting name | Values | Version introduced |
|-----------------------------|---|--------------------------|---------------------------|
| runtimeconfig.json | <code>System.GC.HeapHardLimitSOH</code> | <i>decimal value</i> | .NET 5 |
| Environment variable | <code>COMPlus_GCHeapHardLimitSOH</code> | <i>hexadecimal value</i> | .NET 5 |
| Environment variable | <code>DOTNET_GCHeapHardLimitSOH</code> | <i>hexadecimal value</i> | .NET 6 |

[+] Expand table

| | Setting name | Values | Version introduced |
|-----------------------------|---|--------------------------|---------------------------|
| runtimeconfig.json | <code>System.GC.HeapHardLimitLOH</code> | <i>decimal value</i> | .NET 5 |
| Environment variable | <code>COMPlus_GCHeapHardLimitLOH</code> | <i>hexadecimal value</i> | .NET 5 |
| Environment variable | <code>DOTNET_GCHeapHardLimitLOH</code> | <i>hexadecimal value</i> | .NET 6 |

[+] Expand table

| | Setting name | Values | Version introduced |
|-----------------------------|---|--------------------------|---------------------------|
| runtimeconfig.json | <code>System.GC.HeapHardLimitPOH</code> | <i>decimal value</i> | .NET 5 |
| Environment variable | <code>COMPlus_GCHeapHardLimitPOH</code> | <i>hexadecimal value</i> | .NET 5 |
| Environment variable | <code>DOTNET_GCHeapHardLimitPOH</code> | <i>hexadecimal value</i> | .NET 6 |

These configuration settings don't have specific MSBuild properties. However, you can add a `RuntimeHostConfigurationOption` MSBuild item instead. Use the `runtimeconfig.json` setting name as the value of the `Include` attribute. For an example, see [MSBuild properties](#).

💡 Tip

If you're setting the option in `runtimeconfig.json`, specify a decimal value. If you're setting the option as an environment variable, specify a hexadecimal value. For example, to specify a heap hard limit of 200 mebibytes (MiB), the values would be 209715200 for the JSON file and 0xC800000 or C800000 for the environment variable.

Per-object-heap limit percents

You can specify the GC's allowable heap usage on a per-object-heap basis. The different heaps are the large object heap (LOH), small object heap (SOH), and pinned object heap (POH).

- If you specify a value for any of the `DOTNET_GCHeapHardLimitSOHPercent`, `DOTNET_GCHeapHardLimitLOHPercent`, or `DOTNET_GCHeapHardLimitPOHPercent` settings, you must also specify a value for `DOTNET_GCHeapHardLimitSOHPercent` and `DOTNET_GCHeapHardLimitLOHPercent`. If you don't, the runtime will fail to initialize.
- These settings are ignored if `DOTNET_GCHeapHardLimitSOH`, `DOTNET_GCHeapHardLimitLOH`, and `DOTNET_GCHeapHardLimitPOH` are specified.
- A value of 1 means that GC uses 1% of total physical memory for that object heap.
- Each value must be greater than zero and less than 100. Additionally, the sum of the three percentage values must be less than 100. Otherwise, the runtime will fail to initialize.

[+] Expand table

| Setting name | Values | Version introduced |
|-----------------------------|---|--------------------|
| runtimeconfig.json | <code>System.GC.HeapHardLimitSOHPercent</code> <i>decimal value</i> | .NET 5 |
| Environment variable | <code>COMPlus_GCHeapHardLimitSOHPercent</code> <i>hexadecimal value</i> | .NET 5 |
| Environment variable | <code>DOTNET_GCHeapHardLimitSOHPercent</code> <i>hexadecimal value</i> | .NET 6 |

[+] Expand table

| | Setting name | Values | Version introduced |
|-----------------------------|-----------------------------------|--------------------------|---------------------------|
| runtimeconfig.json | System.GC.HeapHardLimitLOHPercent | <i>decimal value</i> | .NET 5 |
| Environment variable | COMPlus_GCHeapHardLimitLOHPercent | <i>hexadecimal value</i> | .NET 5 |
| Environment variable | DOTNET_GCHeapHardLimitLOHPercent | <i>hexadecimal value</i> | .NET 6 |

[+] Expand table

| | Setting name | Values | Version introduced |
|-----------------------------|-----------------------------------|--------------------------|---------------------------|
| runtimeconfig.json | System.GC.HeapHardLimitPOHPercent | <i>decimal value</i> | .NET 5 |
| Environment variable | COMPlus_GCHeapHardLimitPOHPercent | <i>hexadecimal value</i> | .NET 5 |
| Environment variable | DOTNET_GCHeapHardLimitPOHPercent | <i>hexadecimal value</i> | .NET 6 |

These configuration settings don't have specific MSBuild properties. However, you can add a `RuntimeHostConfigurationOption` MSBuild item instead. Use the *runtimeconfig.json* setting name as the value of the `Include` attribute. For an example, see [MSBuild properties](#).

💡 Tip

If you're setting the option in *runtimeconfig.json*, specify a decimal value. If you're setting the option as an environment variable, specify a hexadecimal value. For example, to limit the heap usage to 30%, the values would be 30 for the JSON file and 0x1E or 1E for the environment variable.

High memory percent

Memory load is indicated by the percentage of physical memory in use. By default, when the physical memory load reaches 90%, garbage collection becomes more aggressive about doing full, compacting garbage collections to avoid paging. When memory load is below 90%, GC favors background collections for full garbage collections, which have shorter pauses but don't reduce the total heap size by much. On

machines with a significant amount of memory (80GB or more), the default load threshold is between 90% and 97%.

The high memory load threshold can be adjusted by the `DOTNET_GCHighMemPercent` environment variable or `System.GC.HighMemoryPercent` JSON configuration setting. Consider adjusting the threshold if you want to control heap size. For example, for the dominant process on a machine with 64GB of memory, it's reasonable for GC to start reacting when there's 10% of memory available. But for smaller processes, for example, a process that only consumes 1GB of memory, GC can comfortably run with less than 10% of memory available. For these smaller processes, consider setting the threshold higher. On the other hand, if you want larger processes to have smaller heap sizes (even when there's plenty of physical memory available), lowering this threshold is an effective way for GC to react sooner to compact the heap down.

 **Note**

For processes running in a container, GC considers the physical memory based on the container limit.

 Expand table

| | Setting name | Values | Version introduced |
|-----------------------------|--|--------------------------|--|
| runtimeconfig.json | <code>System.GC.HighMemoryPercent</code> | <i>decimal value</i> | .NET 5 |
| Environment variable | <code>COMPlus_GCHighMemPercent</code> | <i>hexadecimal value</i> | .NET Core 3.0 .NET Framework 4.7.2 |
| Environment variable | <code>DOTNET_GCHighMemPercent</code> | <i>hexadecimal value</i> | .NET 6 |

This configuration setting doesn't have a specific MSBuild property. However, you can add a `RuntimeHostConfigurationOption` MSBuild item instead. Use the `runtimeconfig.json` setting name as the value of the `Include` attribute. For an example, see [MSBuild properties](#).

 **Tip**

If you're setting the option in `runtimeconfig.json`, specify a decimal value. If you're setting the option as an environment variable, specify a hexadecimal value. For

example, to set the high memory threshold to 75%, the values would be 75 for the JSON file and 0x4B or 4B for the environment variable.

Retain VM

- Configures whether segments that should be deleted are put on a standby list for future use or are released back to the operating system (OS).
- Default: Release segments back to the operating system. This is equivalent to setting the value to `false`.

[Expand table](#)

| | Setting name | Values | Version introduced |
|-----------------------------|--|--|---------------------------|
| runtimconfig.json | <code>System.GC.RetainVM</code> | <code>false</code> - release to OS <code>true</code> - put on standby | .NET Core 1.0 |
| MSBuild property | <code>RetainVMGarbageCollection</code> | <code>false</code> - release to OS <code>true</code> - put on standby | .NET Core 1.0 |
| Environment variable | <code>COMPlus_GCRetainVM</code> | <code>0</code> - release to OS <code>1</code> - put on standby | .NET Core 1.0 |
| Environment variable | <code>DOTNET_GCRetainVM</code> | <code>0</code> - release to OS <code>1</code> - put on standby | .NET 6 |

Examples

runtimconfig.json file:

JSON

```
{  
  "runtimeOptions": {  
    "configProperties": {  
      "System.GC.RetainVM": true  
    }  
  }  
}
```

`runtimconfig.template.json` file:

JSON

```
{  
    "configProperties": {  
        "System.GC.RetainVM": true  
    }  
}
```

Project file:

XML

```
<Project Sdk="Microsoft.NET.Sdk">  
  
    <PropertyGroup>  
        <RetainVMGarbageCollection>true</RetainVMGarbageCollection>  
    </PropertyGroup>  
  
</Project>
```

Large pages

- Specifies whether large pages should be used when a heap hard limit is set.
- Default: Don't use large pages when a heap hard limit is set. This is equivalent to setting the value to `0`.
- This is an experimental setting.

[Expand table](#)

| | Setting name | Values | Version introduced |
|--------------------------------|-----------------------------------|---|--------------------|
| <code>runtimconfig.json</code> | N/A | N/A | N/A |
| Environment variable | <code>COMPlus_GCLargePages</code> | <code>0</code> - disabled <code>1</code> - enabled | .NET Core 3.0 |
| Environment variable | <code>DOTNET_GCLargePages</code> | <code>0</code> - disabled <code>1</code> - enabled | .NET 6 |

Allow large objects

- Configures garbage collector support on 64-bit platforms for arrays that are greater than 2 gigabytes (GB) in total size.
- Default: GC supports arrays greater than 2-GB. This is equivalent to setting the value to 1.
- This option may become obsolete in a future version of .NET.

[\[+\] Expand table](#)

| | Setting name | Values | Version introduced |
|--------------------------------------|--|--|--------------------|
| runtimeconfig.json | N/A | N/A | N/A |
| Environment variable | <code>COMPlus_gcAllowVeryLargeObjects</code> | <input checked="" type="checkbox"/> 1 - enabled <input type="checkbox"/> 0 - disabled | .NET Core 1.0 |
| Environment variable | <code>DOTNET_gcAllowVeryLargeObjects</code> | <input checked="" type="checkbox"/> 1 - enabled <input type="checkbox"/> 0 - disabled | .NET 6 |
| app.config for .NET Framework | <code>gcAllowVeryLargeObjects</code> | <input checked="" type="checkbox"/> 1 - enabled <input type="checkbox"/> 0 - disabled | .NET Framework 4.5 |

Large object heap threshold

- Specifies the threshold size, in bytes, that causes objects to go on the large object heap (LOH).
- The default threshold is 85,000 bytes.
- The value you specify must be larger than the default threshold.
- The value might be capped by the runtime to the maximum possible size for the current configuration. You can inspect the value in use at run time through the [GC.GetConfigurationVariables\(\)](#) API.

[\[+\] Expand table](#)

| | Setting name | Values | Version introduced |
|---------------------------|-------------------------------------|----------------------|--------------------|
| runtimeconfig.json | <code>System.GC.LOHThreshold</code> | <i>decimal value</i> | .NET Core 1.0 |

| Setting name | Values | Version introduced |
|--------------------------------------|---|--------------------|
| Environment variable | <code>COMPlus_GCLOHThreshold</code> <i>hexadecimal value</i> | .NET Core 1.0 |
| Environment variable | <code>DOTNET_GCLOHThreshold</code> <i>hexadecimal value</i> | .NET 6 |
| app.config for .NET Framework | <code>GCLOHThreshold</code> <i>decimal value</i> | .NET Framework 4.8 |

This configuration setting doesn't have a specific MSBuild property. However, you can add a `RuntimeHostConfigurationOption` MSBuild item instead. Use the `runtimeconfig.json` setting name as the value of the `Include` attribute. For an example, see [MSBuild properties](#).

Examples

`runtimeconfig.json` file:

```
JSON
{
  "runtimeOptions": {
    "configProperties": {
      "System.GC.LOHTreshold": 120000
    }
  }
}
```

`runtimeconfig.template.json` file:

```
JSON
{
  "configProperties": {
    "System.GC.LOHTreshold": 120000
  }
}
```

Tip

If you're setting the option in `runtimeconfig.json`, specify a decimal value. If you're setting the option as an environment variable, specify a hexadecimal value. For

example, to set a threshold size of 120,000 bytes, the values would be 120000 for the JSON file and 0x1D4C0 or 1D4C0 for the environment variable.

Standalone GC

- Specifies the name of a GC native library that the runtime loads in place of the default GC implementation. This native library needs to reside in the same directory as the .NET runtime (**coreclr.dll** on Windows, **libcoreclr.so** on Linux).

[Expand table](#)

| | Setting name | Values | Version introduced |
|-----------------------------|-----------------------------|--------------------|--------------------|
| runtimeconfig.json | N/A | N/A | N/A |
| Environment variable | <code>COMPlus_GCName</code> | <i>string_path</i> | .NET Core 2.0 |
| Environment variable | <code>DOTNET_GCName</code> | <i>string_path</i> | .NET 6 |

Conserve memory

- Configures the garbage collector to conserve memory at the expense of more frequent garbage collections and possibly longer pause times.
- Default value is 0 - this implies no change.
- Besides the default value 0, values between 1 and 9 (inclusive) are valid. The higher the value, the more the garbage collector tries to conserve memory and thus to keep the heap small.
- If the value is non-zero, the large object heap will be compacted automatically if it has too much fragmentation.

[Expand table](#)

| | Setting name | Values | Version introduced |
|--------------------------------------|---------------------------------------|--------|--------------------|
| runtimeconfig.json | <code>System.GC.ConserveMemory</code> | 0 - 9 | .NET 6 |
| Environment variable | <code>COMPlus_GCConserveMemory</code> | 0 - 9 | .NET Framework 4.8 |
| Environment variable | <code>DOTNET_GCConserveMemory</code> | 0 - 9 | .NET 6 |
| app.config for .NET Framework | <code>GCConserveMemory</code> | 0 - 9 | .NET Framework 4.8 |

This configuration setting doesn't have a specific MSBuild property. However, you can add a `RuntimeHostConfigurationOption` MSBuild item instead. Use the `runtimeconfig.json` setting name as the value of the `Include` attribute. For an example, see [MSBuild properties](#).

Example `app.config` file:

XML

```
<configuration>
  <runtime>
    <GCConserveMemory enabled="5"/>
  </runtime>
</configuration>
```

Tip

Experiment with different numbers to see which value works best for you. Start with a value between 5 and 7.

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

.NET is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

Runtime configuration options for globalization

Article • 11/11/2023

Invariant mode

- Determines whether a .NET Core app runs in globalization-invariant mode without access to culture-specific data and behavior.
- If you omit this setting, the app runs with access to cultural data. This is equivalent to setting the value to `false`.
- For more information, see [.NET Core globalization invariant mode ↗](#).

| | Setting name | Values |
|-----------------------------|--|---|
| runtimesconfig.json | <code>System.Globalization.Invariant</code> | <code>false</code> - access to cultural data <code>true</code> - run in invariant mode |
| MSBuild property | <code>InvariantGlobalization</code> | <code>false</code> - access to cultural data <code>true</code> - run in invariant mode |
| Environment variable | <code>DOTNET_SYSTEM_GLOBALIZATION_INVARIANT</code> | <code>0</code> - access to cultural data <code>1</code> - run in invariant mode |

Examples

runtimesconfig.json file:

```
JSON

{
  "runtimeOptions": {
    "configProperties": {
      "System.Globalization.Invariant": true
    }
  }
}
```

runtimesconfig.template.json file:

```
JSON

{
  "configProperties": {
    "System.Globalization.Invariant": true
  }
}
```

```
}
```

Project file:

XML

```
<Project Sdk="Microsoft.NET.Sdk">

<PropertyGroup>
    <InvariantGlobalization>true</InvariantGlobalization>
</PropertyGroup>

</Project>
```

Era year ranges

- Determines whether range checks for calendars that support multiple eras are relaxed or whether dates that overflow an era's date range throw an [ArgumentException](#).
- If you omit this setting, range checks are relaxed. This is equivalent to setting the value to `false`.
- For more information, see [Calendars, eras, and date ranges: Relaxed range checks](#).

| Setting name | Values |
|-----------------------------|--|
| runtimeconfig.json | <code>Switch.System.Globalization.EnforceJapaneseEraYearRanges</code> <code>false</code> - relaxed range checks <code>true</code> - overflows cause an exception |
| Environment variable | N/A |

This configuration setting doesn't have a specific MSBuild property. However, you can add a `RuntimeHostConfigurationOption` MSBuild item instead. Use the `runtimeconfig.json` setting name as the value of the `Include` attribute. For an example, see [MSBuild properties](#).

Japanese date parsing

- Determines whether a string that contains either "1" or "Gannen" as the year parses successfully or whether only "1" is supported.

- If you omit this setting, strings that contain either "1" or "Gannen" as the year parse successfully. This is equivalent to setting the value to `false`.
- For more information, see [Represent dates in calendars with multiple eras](#).

| | Setting name | Values |
|-----------------------------|---|--|
| runtimeconfig.json | <code>Switch.System.Globalization.EnforceLegacyJapaneseDateParsing</code> | <code>false</code> - "Gannen" or "1" is supported <code>true</code> - only "1" is supported |
| Environment variable | N/A | N/A |

This configuration setting doesn't have a specific MSBuild property. However, you can add a `RuntimeHostConfigurationOption` MSBuild item instead. Use the `runtimeconfig.json` setting name as the value of the `Include` attribute. For an example, see [MSBuild properties](#).

Japanese year format

- Determines whether the first year of a Japanese calendar era is formatted as "Gannen" or as a number.
- If you omit this setting, the first year is formatted as "Gannen". This is equivalent to setting the value to `false`.
- For more information, see [Represent dates in calendars with multiple eras](#).

| | Setting name | Values |
|-----------------------------|---|---|
| runtimeconfig.json | <code>Switch.System.Globalization.FormatJapaneseFirstYearAsANumber</code> | <code>false</code> - format as "Gannen" <code>true</code> - format as number |
| Environment variable | N/A | N/A |

This configuration setting doesn't have a specific MSBuild property. However, you can add a `RuntimeHostConfigurationOption` MSBuild item instead. Use the `runtimeconfig.json` setting name as the value of the `Include` attribute. For an example, see [MSBuild properties](#).

NLS

- Determines whether .NET uses National Language Support (NLS) or International Components for Unicode (ICU) globalization APIs for Windows apps. .NET 5 and later versions use ICU globalization APIs by default on Windows 10 May 2019 Update and later versions.
- If you omit this setting, .NET uses ICU globalization APIs by default. This is equivalent to setting the value to `false`.
- For more information, see [Globalization APIs use ICU libraries on Windows](#).

| | Setting name | Values | Introduced |
|-----------------------------|---|---|-------------------|
| runtimesconfig.json | <code>System.Globalization.UseNls</code> | <code>false</code> - Use ICU globalization APIs <code>true</code> - Use NLS globalization APIs | .NET 5 |
| Environment variable | <code>DOTNET_SYSTEM_GLOBALIZATION_USENLS</code> | <code>false</code> - Use ICU globalization APIs <code>true</code> - Use NLS globalization APIs | .NET 5 |

This configuration setting doesn't have a specific MSBuild property. However, you can add a `RuntimeHostConfigurationOption` MSBuild item instead. Use the `runtimesconfig.json` setting name as the value of the `Include` attribute. For an example, see [MSBuild properties](#).

Predefined cultures

- Configures whether apps can create cultures other than the invariant culture when [globalization-invariant mode](#) is enabled.
- If you omit this setting, .NET restricts the creation of cultures in globalization-invariant mode. This is equivalent to setting the value to `true`.
- For more information, see [Culture creation and case mapping in globalization-invariant mode](#).

| | Setting name | Values | Introduced |
|----------------------------|--|---|-------------------|
| runtimesconfig.json | <code>System.Globalization.PredefinedCulturesOnly</code> | <code>true</code> - In globalization-invariant mode, don't allow creation of any culture except the | .NET 6 |

| Setting name | Values | Introduced |
|---|---|------------|
| | invariant culture. <code>false</code> - Allow creation of any culture. | |
| MSBuild property <code>PredefinedCulturesOnly</code> | <code>true</code> - In globalization-invariant mode, don't allow creation of any culture except the invariant culture. <code>false</code> - Allow creation of any culture. | .NET 6 |
| Environment variable <code>DOTNET_SYSTEM_GLOBALIZATION_PREDEFINED_CULTURES_ONLY</code> | <code>true</code> - In globalization-invariant mode, don't allow creation of any culture except the invariant culture. <code>false</code> - Allow creation of any culture. | .NET 6 |

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



.NET feedback

The .NET documentation is open source. Provide feedback here.

[Open a documentation issue](#)

[Provide product feedback](#)

Runtime configuration options for networking

Article • 11/11/2023

HTTP/2 protocol

- Configures whether support for the HTTP/2 protocol is enabled.
- Introduced in .NET Core 3.0.
- .NET Core 3.0 only: If you omit this setting, support for the HTTP/2 protocol is disabled. This is equivalent to setting the value to `false`.
- .NET Core 3.1 and .NET 5+: If you omit this setting, support for the HTTP/2 protocol is enabled. This is equivalent to setting the value to `true`.

| | Setting name | Values |
|-----------------------------|---|--|
| runtimesconfig.json | <code>System.Net.Http.SocketsHttpHandler.Http2Support</code> | <code>false</code> - disabled <code>true</code> - enabled |
| Environment variable | <code>DOTNET_SYSTEM_NET_HTTP_SOCKETSHANDLER_HTTP2SUPPORT</code> | <code>0</code> - disabled <code>1</code> - enabled |

This configuration setting doesn't have a specific MSBuild property. However, you can add a `RuntimeHostConfigurationOption` MSBuild item instead. Use the `runtimesconfig.json` setting name as the value of the `Include` attribute. For an example, see [MSBuild properties](#).

SPN creation in HttpClient (.NET 6 and later)

- Impacts generation of [service principal names](#) (SPN) for Kerberos and NTLM authentication when `Host` header is missing and target is not running on default port.
- .NET Core 2.x and 3.x do not include port in SPN.
- .NET Core 5.x does include port in SPN
- .NET 6 and later versions don't include the port, but the behavior is configurable.

| Setting name | Values |
|-----------------------------|--|
| runtimeconfig.json | <code>System.Net.Http.UsePortInSpn</code> <code>true</code> - includes port number in SPN, for example, <code>HTTP/host:port</code> <code>false</code> - does not include port in SPN, for example, <code>HTTP/host</code> |
| Environment variable | <code>DOTNET_SYSTEM_NET_HTTP_USEPORTINSPN</code> <code>1</code> - includes port number in SPN, for example, <code>HTTP/host:port</code> <code>0</code> - does not include port in SPN, for example, <code>HTTP/host</code> |

This configuration setting doesn't have a specific MSBuild property. However, you can add a `RuntimeHostConfigurationOption` MSBuild item instead. Use the *runtimeconfig.json* setting name as the value of the `Include` attribute. For an example, see [MSBuild properties](#).

UseSocketsHttpHandler (.NET Core 2.1-3.1 only)

- Configures whether `System.Net.Http.HttpClientHandler` uses `System.Net.Http.SocketsHttpHandler` or older HTTP protocol stacks (`WinHttpHandler` on Windows and `CurlHandler`, an internal class implemented on top of `libcurl`, on Linux).

! Note

You may be using high-level networking APIs instead of directly instantiating the `HttpClientHandler` class. This setting also affects which HTTP protocol stack is used by high-level networking APIs, including `HttpClient` and `HttpClientFactory`.

- If you omit this setting, `HttpClientHandler` uses `SocketsHttpHandler`. This is equivalent to setting the value to `true`.

| Setting name | Values |
|---------------------------|---|
| runtimeconfig.json | <code>System.Net.Http.UseSocketsHttpHandler</code> <code>true</code> - enables the use of <code>SocketsHttpHandler</code> <code>false</code> - enables the use of <code>WinHttpHandler</code> on Windows or <code>libcurl</code> on Linux |

| Setting name | Values |
|-----------------------------|--|
| Environment variable | <code>DOTNET_SYSTEM_NET_HTTP_USESOCKETSHTTPHANDLER</code> 1 - enables the use of SocketsHttpHandler 0 - enables the use of WinHttpHandler on Windows or libcurl on Linux |

ⓘ Note

Starting in .NET 5, the `System.Net.Http.UseSocketsHttpHandler` setting is no longer available.

Latin1 headers (.NET Core 3.1 only)

- This switch allows relaxing the HTTP header validation, enabling [SocketsHttpHandler](#) to send ISO-8859-1 (Latin-1) encoded characters in headers.
- If you omit this setting, an attempt to send a non-ASCII character will result in [HttpRequestException](#). This is equivalent to setting the value to `false`.

| Setting name | Values |
|-----------------------------|--|
| runtimeconfig.json | <code>System.Net.Http.SocketsHttpHandler.AllowLatin1Headers</code> <code>false</code> - disabled <code>true</code> - enabled |
| Environment variable | <code>DOTNET_SYSTEM_NET_HTTP_SOCKETSHTTPHANDLER_ALLOWLATIN1HEADERS</code> <code>0</code> - disabled <code>1</code> - enabled |

ⓘ Note

This option is only available in .NET Core 3.1 since version 3.1.9, and not in previous or later versions. In .NET 5 we recommend using [RequestHeaderEncodingSelector](#).

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

Runtime configuration options for threading

Article • 09/23/2023

This article details the settings you can use to configure threading in .NET.

! Note

.NET 6 standardizes on the prefix `DOTNET_` instead of `COMPlus_` for environment variables that configure .NET run-time behavior. However, the `COMPlus_` prefix will continue to work. If you're using a previous version of the .NET runtime, you should still use the `COMPlus_` prefix for environment variables.

Use all CPU groups on Windows

- On machines that have multiple CPU groups, this setting configures whether components such as the thread pool use all CPU groups or only the primary CPU group of the process. The setting also affects what `Environment.ProcessorCount` returns.
- When this setting is enabled, all CPU groups are used and threads are also [automatically distributed across CPU groups](#) by default.
- This setting is enabled by default on Windows 11 and later versions, and disabled by default on Windows 10 and earlier versions. For this setting to take effect when enabled, the GC must also be configured to use all CPU groups; for more information, see [GC CPU groups](#).

| | Setting name | Values |
|-----------------------------|---|--|
| runtimesconfig.json | N/A | N/A |
| Environment variable | <code>COMPlus_Thread_UseAllCpuGroups</code> or <code>DOTNET_Thread_UseAllCpuGroups</code> | <input type="radio"/> 0 - disabled <input checked="" type="radio"/> 1 - enabled |

Assign threads to CPU groups on Windows

- On machines that have multiple CPU groups and [all CPU groups are being used](#), this setting configures whether threads are automatically distributed across CPU groups.
- When this setting is enabled, new threads are assigned to a CPU group in a way that tries to fully populate a CPU group that is already in use before utilizing a new CPU group.
- This setting is enabled by default.

| | Setting name | Values |
|-----------------------------|---|--|
| runtimesconfig.json | N/A | N/A |
| Environment variable | <code>COMPlus_Thread_AssignCpuGroups</code> or <code>DOTNET_Thread_AssignCpuGroups</code> | <input type="radio"/> 0 - disabled <input checked="" type="radio"/> 1 - enabled |

Minimum threads

- Specifies the minimum number of threads for the worker thread pool.
- Corresponds to the [ThreadPool.SetMinThreads](#) method.

| | Setting name | Values |
|-----------------------------|---|--|
| runtimeconfig.json | <code>System.Threading.ThreadPool.MinThreads</code> | An integer that represents the minimum number of threads |
| MSBuild property | <code>ThreadPoolMinThreads</code> | An integer that represents the minimum number of threads |
| Environment variable | N/A | N/A |

Examples

runtimeconfig.json file:

```
JSON

{
  "runtimeOptions": {
    "configProperties": {
      "System.Threading.ThreadPool.MinThreads": 4
    }
  }
}
```

runtimeconfig.template.json file:

```
JSON

{
  "configProperties": {
    "System.Threading.ThreadPool.MinThreads": 4
  }
}
```

Project file:

```
XML

<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <ThreadPoolMinThreads>4</ThreadPoolMinThreads>
  </PropertyGroup>

</Project>
```

Maximum threads

- Specifies the maximum number of threads for the worker thread pool.
- Corresponds to the [ThreadPool.SetMaxThreads](#) method.

| Setting name | Values |
|-----------------------------|---|
| runtimeconfig.json | <code>System.Threading.ThreadPool.MaxThreads</code> An integer that represents the maximum number of threads |
| MSBuild property | <code>ThreadPoolMaxThreads</code> An integer that represents the maximum number of threads |
| Environment variable | N/A |

Examples

runtimeconfig.json file:

JSON

```
{
  "runtimeOptions": {
    "configProperties": {
      "System.Threading.ThreadPool.MaxThreads": 20
    }
  }
}
```

runtimeconfig.template.json file:

JSON

```
{
  "configProperties": {
    "System.Threading.ThreadPool.MaxThreads": 20
  }
}
```

Project file:

XML

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <ThreadPoolMaxThreads>20</ThreadPoolMaxThreads>
  </PropertyGroup>

</Project>
```

Windows thread pool

- For projects on Windows, configures whether thread pool thread management is delegated to the Windows thread pool.
- If you omit this setting or the platform is not Windows, the .NET thread pool is used instead.

- Only applications published with Native AOT on Windows use the Windows thread pool by default, for which you can opt to use the .NET thread pool instead by disabling the config setting.
- The Windows thread pool may perform better in some cases, such as in cases where the minimum number of threads is configured to a high value, or when the Windows thread pool is already being heavily used by the app. There may also be cases where the .NET thread pool performs better, such as in heavy I/O handling on larger machines. It's advisable to check performance metrics when changing this config setting.
- Some APIs are not supported when using the Windows thread pool, such as [ThreadPool.SetMinThreads](#), [ThreadPool.SetMaxThreads](#), and [ThreadPool.BindHandle\(SafeHandle\)](#). Thread pool config settings for minimum and maximum threads are also not effective. An alternative to [ThreadPool.BindHandle\(SafeHandle\)](#) is the [ThreadPoolBoundHandle](#) class.

| Setting name | Values | Version introduced |
|-----------------------------|---|--------------------|
| runtimesconfig.json | <code>System.Threading.ThreadPool.UseWindowsThreadPool</code> <code>true</code> - enabled <code>false</code> - disabled | .NET 8 |
| MSBuild property | <code>UseWindowsThreadPool</code> <code>true</code> - enabled <code>false</code> - disabled | .NET 8 |
| Environment variable | <code>DOTNET_Threadpool_UseWindowsThreadPool</code> <code>1</code> - enabled <code>0</code> - disabled | .NET 8 |

Examples

runtimesconfig.json file:

```
JSON

{
  "runtimeOptions": {
    "configProperties": {
      "System.Threading.ThreadPool.UseWindowsThreadPool": true
    }
  }
}
```

runtimesconfig.template.json file:

```
JSON

{
  "configProperties": {
    "System.Threading.ThreadPool.UseWindowsThreadPool": true
  }
}
```

Project file:

```
XML
```

```

<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <UseWindowsThreadPool>true</UseWindowsThreadPool>
  </PropertyGroup>

</Project>

```

Thread injection in response to blocking work items

In some cases, the thread pool detects work items that block its threads. To compensate, it injects more threads. In .NET 6+, you can use the following [runtime configuration](#) settings to configure thread injection in response to blocking work items. Currently, these settings take effect only for work items that wait for another task to complete, such as in typical [sync-over-async](#) cases.

| runtimeconfig.json setting name | Description | Version introduced |
|--|---|---------------------------|
| <code>System.Threading.ThreadPool.Blocking.ThreadsToAddWithoutDelay_ProcCountFactor</code> | After the thread count based on <code>MinThreads</code> is reached, this value (after it is multiplied by the processor count) specifies how many additional threads may be created without a delay. | .NET 6 |
| <code>System.Threading.ThreadPool.Blocking.ThreadsPerDelayStep_ProcCountFactor</code> | After the thread count based on <code>ThreadsToAddWithoutDelay</code> is reached, this value (after it is multiplied by the processor count) specifies after how many threads an additional <code>DelayStepMs</code> would be added to the delay before each new thread is created. | .NET 6 |
| <code>System.Threading.ThreadPool.Blocking.DelayStepMs</code> | After the thread count based on <code>ThreadsToAddWithoutDelay</code> is reached, this value specifies how much additional delay to add per <code>ThreadsPerDelayStep</code> threads, which would be applied before each new thread is created. | .NET 6 |
| <code>System.Threading.ThreadPool.Blocking.MaxDelayMs</code> | After the thread count based on <code>ThreadsToAddWithoutDelay</code> is reached, this value | .NET 6 |

| runtimesconfig.json setting name | Description | Version introduced |
|---|---|---------------------------|
| <code>System.Threading.ThreadPool.Blocking.IgnoreMemoryUsage</code> | specifies the max delay to use before each new thread is created. | |

How the configuration settings take effect

- After the thread count based on `MinThreads` is reached, up to `ThreadsToAddWithoutDelay` additional threads may be created without a delay.
- After that, before each additional thread is created, a delay is induced, starting with `DelayStepMs`.
- For every `ThreadsPerDelayStep` threads that are added with a delay, an additional `DelayStepMs` is added to the delay.
- The delay may not exceed `MaxDelayMs`.
- Delays are only induced before creating threads. If threads are already available, they would be released without delay to compensate for blocking work items.
- Physical memory usage and limits are also used and, beyond a threshold, the system switches to slower thread injection.

Examples

runtimesconfig.json file:

JSON

```
{
  "runtimeOptions": {
    "configProperties": {
      "System.Threading.ThreadPool.Blocking.ThreadsToAddWithoutDelay_ProcCountFactor": 5
    }
  }
}
```

runtimesconfig.template.json file:

JSON

```
{
  "configProperties": {
    "System.Threading.ThreadPool.Blocking.ThreadsToAddWithoutDelay_ProcCountFactor": 5
  }
}
```

AutoreleasePool for managed threads

This option configures whether each managed thread receives an implicit [NSAutoreleasePool](#) when running on a supported macOS platform.

| | Setting name | Values | Version introduced |
|-----------------------------|--|---------------|---------------------------|
| runtimeconfig.json | <code>System.Threading.Thread.EnableAutoreleasePool</code> | true or false | .NET 6 |
| MSBuild property | <code>AutoreleasePoolSupport</code> | true or false | .NET 6 |
| Environment variable | N/A | N/A | N/A |

Examples

runtimeconfig.json file:

JSON

```
{
  "runtimeOptions": {
    "configProperties": {
      "System.Threading.Thread.EnableAutoreleasePool": true
    }
  }
}
```

runtimeconfig.template.json file:

JSON

```
{
  "configProperties": {
    "System.Threading.Thread.EnableAutoreleasePool": true
  }
}
```

Project file:

XML

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <AutoreleasePoolSupport>true</AutoreleasePoolSupport>
  </PropertyGroup>
```

 Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

 .NET feedback

The .NET documentation is open source. Provide feedback here.

 Open a documentation issue

 Provide product feedback

Runtime configuration options for WPF

Article • 11/11/2023

This article details the settings you can use to configure Windows Presentation Framework (WPF) in .NET.

⚠ Note

.NET 6 standardizes on the prefix `DOTNET_` instead of `COMPlus_` for environment variables that configure .NET run-time behavior. However, the `COMPlus_` prefix will continue to work. If you're using a previous version of the .NET runtime, you should still use the `COMPlus_` prefix for environment variables.

Hardware acceleration in RDP

- Configures whether hardware acceleration is used for WPF apps that are accessed through Remote Desktop Protocol (RDP). Hardware acceleration refers to the use of a computer's graphics processing unit (GPU) to speed up the rendering of graphics and visual effects in an application. This can result in improved performance and more seamless, responsive graphics.
- If you omit this setting, graphics are rendered by software instead. This is equivalent to setting the value to `false`.

| Setting name | Values | Version introduced |
|-----------------------------------|--|--------------------|
| <code>runtimconfig.json</code> | <code>Switch.System.Windows.Media.EnableHardwareAccelerationInRdp</code> <code>true</code> - enabled <code>false</code> - disabled | .NET 8 |
| <code>Environment variable</code> | N/A | N/A |

This configuration setting doesn't have a specific MSBuild property. However, you can add a `RuntimeHostConfigurationOption` MSBuild item instead. Use the `runtimconfig.json` setting name as the value of the `Include` attribute. For an example, see [MSBuild properties](#).

Collaborate with us on
GitHub

.NET .NET feedback

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

The .NET documentation is open source.
Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

Troubleshoot app launch failures

Article • 04/01/2023

This article describes some common reasons and possible solutions for application launch failures. It relates to [framework-dependent applications](#), which rely on a .NET installation on your machine.

If you already know which .NET version you need, you can download it from [.NET downloads](#).

.NET installation not found

If a .NET installation isn't found, the application fails to launch with a message similar to:

```
Console

You must install .NET to run this application.

App: C:\repos\myapp\myapp.exe
Architecture: x64
Host version: 7.0.0
.NET location: Not found
```

The error message includes a link to download .NET. You can follow that link to get to the appropriate download page. You can also pick the .NET version (specified by `Host version`) from [.NET downloads](#).

On the [download page](#) for the required .NET version, find the **.NET Runtime** download that matches the architecture listed in the error message. You can then install it by downloading and running an **Installer**.

Alternatively, on the [download page](#) for the required .NET version, you can download **Binaries** for the specified architecture.

Required framework not found

If a required framework or compatible version isn't found, the application fails to launch with a message similar to:

```
Console
```

You must install or update .NET to run this application.

App: C:\repos\myapp\myapp.exe

Architecture: x64

Framework: 'Microsoft.NETCore.App', version '5.0.15' (x64)

.NET location: C:\Program Files\dotnet\

The following frameworks were found:

6.0.2 at [c:\Program Files\dotnet\shared\Microsoft.NETCore.App]

The error indicates the name, version, and architecture of the missing framework and the location at which it's expected to be installed. To run the application, you can [install a compatible runtime](#) at the specified ".NET location". If the application targets a lower version than one you have installed and you'd like to run it on a higher version, you can also [configure roll-forward behavior](#) for the application.

Install a compatible runtime

The error message includes a link to download the missing framework. You can follow this link to get to the appropriate download page.

Alternately, you can download a runtime from the [.NET downloads](#) page. There are multiple .NET runtime downloads.

The following table shows the frameworks that each runtime contains.

[+] Expand table

| Runtime download | Included frameworks |
|----------------------|---|
| ASP.NET Core Runtime | Microsoft.NETCore.App Microsoft.AspNetCore.App |
| .NET Desktop Runtime | Microsoft.NETCore.App Microsoft.WindowsDesktop.App |
| .NET Runtime | Microsoft.NETCore.App |

Select a runtime download that contains the missing framework, and then install it.

On the [download page](#) for the required .NET version, find the runtime download that matches the architecture listed in the error message. You likely want to download an [Installer](#).

Alternatively, on the [download page](#) for the required .NET version, you can download **Binaries** for the specified architecture.

In most cases, when the application that failed to launch is using such an installation, the ".NET location" in the error message points to:

```
%ProgramFiles%\dotnet
```

Other options

There are other installation and workaround options to consider.

Run the dotnet-install script

Download the [dotnet-install script](#) for your operating system. Run the script with options based on the information in the error message. The [dotnet-install script reference page](#) shows all available options.

Launch [PowerShell](#) and run:

```
PowerShell  
  
dotnet-install.ps1 -Architecture <architecture> -InstallDir <directory> -  
Runtime <runtime> -Version <version>
```

For example, the error message in the previous section would correspond to:

```
PowerShell  
  
dotnet-install.ps1 -Architecture x64 -InstallDir "C:\Program Files\dotnet\"  
-Runtime dotnet -Version 5.0.15
```

If you encounter an error stating that running scripts is disabled, you may need to set the [execution policy](#) to allow the script to run:

```
PowerShell  
  
Set-ExecutionPolicy Bypass -Scope Process
```

For more information on installation using the script, see [Install with PowerShell automation](#).

Download binaries

You can download a binary archive of .NET from the [download page](#). From the **Binaries** column of the runtime download, download the binary release matching the required architecture. Extract the downloaded archive to the ".NET location" specified in the error message.

For more information about manual installation, see [Install .NET on Windows](#)

Configure roll-forward behavior

If you already have a higher version of the required framework installed, you can make the application run on that higher version by configuring its roll-forward behavior.

When running the application, you can specify the [--roll-forward command line option](#) or set the [DOTNET_ROLL_FORWARD environment variable](#). By default, an application requires a framework that matches the same major version that the application targets, but can use a higher minor or patch version. However, application developers may have specified a different behavior. For more information, see [Framework-dependent apps roll-forward](#).

ⓘ Note

Since using this option lets the application run on a different framework version than the one for which it was designed, it may result in unintended behavior due to changes between versions of a framework.

Breaking changes

Multi-level lookup disabled for .NET 7 and later

On Windows, before .NET 7, the application could search for frameworks in multiple [install locations](#).

1. Subdirectories relative to:

- `dotnet` executable when running the application through `dotnet`.
- `DOTNET_ROOT` environment variable (if set) when running the application through its executable (`apphost`).

2. Globally registered install location (if set) in

`HKLM\SOFTWARE\dotnet\Setup\InstalledVersions\<arch>\InstallLocation`.

3. Default install location of `%ProgramFiles%\dotnet` (or `%ProgramFiles(x86)%\dotnet` for 32-bit processes on 64-bit Windows).

This multi-level lookup behavior was enabled by default but could be disabled by setting the environment variable `DOTNET_MULTILEVEL_LOOKUP=0`.

For applications targeting .NET 7 and later, multi-level lookup is completely disabled and only one location—the first location where a .NET installation is found—is searched. When an application is run through `dotnet`, frameworks are only searched for in subdirectories relative to `dotnet`. When an application is run through its executable (`apphost`), frameworks are only searched for in the first of the previously listed locations where .NET is found.

For more information, see [Multi-level lookup is disabled](#).

See also

- [Install .NET](#)
- [.NET install locations ↗](#)
- [Check installed .NET versions](#)
- [Framework-dependent applications](#)



Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



.NET feedback

.NET is an open source project.
Select a link to provide feedback:

[Open a documentation issue](#)

[Provide product feedback](#)