**Contents**

[.NET Stack 10](#_Toc95305137)

[The history behind .NET: from the .NET Framework first release to .NET 6 10](#_Toc95305138)

[The .NET Framework era 10](#_Toc95305139)

[The .NET CORE era 10](#_Toc95305140)

[The .NET 5 and .NET 6 era 11](#_Toc95305141)

[What is .NET development platform 14](#_Toc95305142)

[.NET framework 14](#_Toc95305143)

[.NET Core 14](#_Toc95305144)

[Xamarin 15](#_Toc95305145)

[Universal Windows Platform (UWP) 15](#_Toc95305146)

[.NET Standard 15](#_Toc95305147)

[Common language runtime (CLR) 16](#_Toc95305148)

[Supported languages 16](#_Toc95305149)

[The Framework 17](#_Toc95305150)

[Stack vs Heap 19](#_Toc95305151)

[C# - Input / Output 22](#_Toc95305152)

[C# Output 22](#_Toc95305153)

[Difference between WriteLine() and Write() method 22](#_Toc95305154)

[Printing Variables and Literals using WriteLine() and Write() 23](#_Toc95305155)

[Combining (Concatenating) two strings using + operator and printing them 24](#_Toc95305156)

[Printing concatenated string using Formatted String [Better Alternative] 24](#_Toc95305157)

[C# Input 25](#_Toc95305158)

[Difference between ReadLine(), Read() and ReadKey() method: 25](#_Toc95305159)

[Reading numeric values (integer and floating point types) 26](#_Toc95305160)

[C# - Data Types (Primitive) 28](#_Toc95305161)

[Predefined Data Types in C# 28](#_Toc95305162)

[Alias vs .NET Type 29](#_Toc95305163)

[Default Values 30](#_Toc95305164)

[Conversions 30](#_Toc95305165)

[Enums 31](#_Toc95305166)

[Enum Values 32](#_Toc95305167)

[Access an Enum 33](#_Toc95305168)

[Conversion 33](#_Toc95305169)

[C# - Arrays 35](#_Toc95305170)

[Array Declaration and Initialization 35](#_Toc95305171)

[Accessing Array Elements 36](#_Toc95305172)

[Accessing Array using for Loop 36](#_Toc95305173)

[Accessing Array using foreach Loop 37](#_Toc95305174)

[LINQ Methods 37](#_Toc95305175)

[Passing Array as Argument 37](#_Toc95305176)

[C# - Multidimensional Arrays 39](#_Toc95305177)

[C# - Jagged Arrays: An Array of Array 42](#_Toc95305178)

[C# - Collections 45](#_Toc95305179)

[Generic Collections 45](#_Toc95305180)

[Non-generic Collections 45](#_Toc95305181)

[C# - Array List 46](#_Toc95305182)

[Create an ArrayList 46](#_Toc95305183)

[Adding Elements in ArrayList 46](#_Toc95305184)

[Accessing an ArrayList 47](#_Toc95305185)

[Iterate an ArrayList 48](#_Toc95305186)

[Insert Elements in ArrayList 48](#_Toc95305187)

[Remove Elements from ArrayList 49](#_Toc95305188)

[Check Element in ArrayList 50](#_Toc95305189)

[ArrayList Class 50](#_Toc95305190)

[ArrayList Properties 51](#_Toc95305191)

[ArrayList Methods 52](#_Toc95305192)

[C# - List 52](#_Toc95305193)

[List<T> Characteristics 53](#_Toc95305194)

[Creating a List 53](#_Toc95305195)

[Adding an Array in a List 54](#_Toc95305196)

[Accessing a List 54](#_Toc95305197)

[Accessing a List using LINQ 54](#_Toc95305198)

[Insert Elements in List 55](#_Toc95305199)

[Remove Elements from List 55](#_Toc95305200)

[Check Elements in List 56](#_Toc95305201)

[List<T> Class Hierarchy 56](#_Toc95305202)

[List<T> Class Properties and Methods 57](#_Toc95305203)

[C# - SortedList<TKey, TValue> 58](#_Toc95305204)

[SortedList Characteristics 59](#_Toc95305205)

[Creating a SortedList 59](#_Toc95305206)

[Accessing SortedList 61](#_Toc95305207)

[Remove Elements from SortedList 62](#_Toc95305208)

[SortedList Class Hierarchy 63](#_Toc95305209)

[C# - Dictionary<TKey, TValue> 63](#_Toc95305210)

[Dictionary Characteristics 63](#_Toc95305211)

[Creating a Dictionary 64](#_Toc95305212)

[Access Dictionary Elements 64](#_Toc95305213)

[Update Dictionary 65](#_Toc95305214)

[Remove Elements in Dictionary 65](#_Toc95305215)

[Dictionary Class Hierarchy 66](#_Toc95305216)

[C# - Hashtable 67](#_Toc95305217)

[Hashtable Characteristics 67](#_Toc95305218)

[Creating a Hashtable 67](#_Toc95305219)

[Update Hashtable 68](#_Toc95305220)

[Remove Elements in Hashtable 68](#_Toc95305221)

[Hashtable Class Hierarchy 69](#_Toc95305222)

[C# - Stack<T> 71](#_Toc95305223)

[Stack<T> Characteristics 71](#_Toc95305224)

[Creating a Stack 71](#_Toc95305225)

[Stack<T> Properties and Methods: 72](#_Toc95305226)

[Pop() 72](#_Toc95305227)

[Peek() 72](#_Toc95305228)

[Contains() 73](#_Toc95305229)

[C# - Queue<T> 73](#_Toc95305230)

[Queue<T> Characteristics 73](#_Toc95305231)

[Creating a Queue 74](#_Toc95305232)

[Queue<T> Properties and Methods 74](#_Toc95305233)

[Retrieve Elements from a Queue 75](#_Toc95305234)

[Contains() 75](#_Toc95305235)

[C# - HashSet 76](#_Toc95305236)

[How to create a HashSet? 76](#_Toc95305237)

[How to remove elements from the HashSet? 78](#_Toc95305238)

[Set Operations 79](#_Toc95305239)

[C# - Type Conversions (Type Casting) 84](#_Toc95305240)

[Implicit Type Casting / Automatic Type Conversion 84](#_Toc95305241)

[Explicit Type Casting 85](#_Toc95305242)

[C# - Serialization 88](#_Toc95305243)

[How serialization works 88](#_Toc95305244)

[Namespaces involved 88](#_Toc95305245)

[Uses for serialization 89](#_Toc95305246)

[JSON serialization 89](#_Toc95305247)

[Binary and XML serialization 89](#_Toc95305248)

[Making an object serializable 89](#_Toc95305249)

[Basic and custom serialization 90](#_Toc95305250)

[Example 1: Binary Serialization 90](#_Toc95305251)

[Explanation 91](#_Toc95305252)

[Example 2: Class / Object Serialization 91](#_Toc95305253)

[Example 3: Array Serialization 93](#_Toc95305254)

[Example 4: Serialize Class / Object 94](#_Toc95305255)

[Code Explanation: 95](#_Toc95305256)

[Serialize and Deserialize JSON 95](#_Toc95305257)

[Namespaces 96](#_Toc95305258)

[Serialize (Write) .NET objects as JSON 96](#_Toc95305259)

[JSON Serialization 96](#_Toc95305260)

[Serialize JSON to a File 97](#_Toc95305261)

[Serialize JSON to a File Asynchronously 98](#_Toc95305262)

[Serialize JSON using Generics 98](#_Toc95305263)

[Serialization of Class with Collections and User Defined Types 99](#_Toc95305264)

[Deserialize (Read) JSON as .NET Objects 101](#_Toc95305265)

[Deserialize a JSON String 101](#_Toc95305266)

[Deserialize JSON from a File 102](#_Toc95305267)

[Deserialize JSON from a File Asynchronously 103](#_Toc95305268)

[Serialize to Formatted JSON 104](#_Toc95305269)

[Include fields 105](#_Toc95305270)

[HttpClient and HttpContent extension methods 107](#_Toc95305271)

[Serialize and Deserialize XML 108](#_Toc95305272)

[Serialize (Read) Data from an XML File 108](#_Toc95305273)

[Deserialize (Write) Data to an XML File 109](#_Toc95305274)

[C# - OOP 121](#_Toc95305275)

[OOP Principles 121](#_Toc95305276)

[C# Class Members 121](#_Toc95305277)

[Indexers 121](#_Toc95305278)

[Difference between Indexers and Properties 123](#_Toc95305279)

[C# - Access Modifiers 124](#_Toc95305280)

[protected 124](#_Toc95305281)

[internal 125](#_Toc95305282)

[C# - Modifiers 127](#_Toc95305283)

[Override 127](#_Toc95305284)

[Readonly 129](#_Toc95305285)

[Static 131](#_Toc95305286)

[Virtual 134](#_Toc95305287)

[C# - Data Types 138](#_Toc95305288)

[Anonymous Types 138](#_Toc95305289)

[Dynamic Types 140](#_Toc95305290)

[Methods and Parameters 141](#_Toc95305291)

[Nullable Types 142](#_Toc95305292)

[Shorthand Syntax for Nullable Types 144](#_Toc95305293)

[?? Operator 144](#_Toc95305294)

[Assignment Rules 144](#_Toc95305295)

[Value Type vs Reference Type 145](#_Toc95305296)

[Value Type 145](#_Toc95305297)

[Passing Value Type Variables 146](#_Toc95305298)

[Reference Type 146](#_Toc95305299)

[Passing Reference Type Variables 147](#_Toc95305300)

[Null 148](#_Toc95305301)

[Struct 149](#_Toc95305302)

[Structure Declaration 149](#_Toc95305303)

[Constructors in Structure 150](#_Toc95305304)

[Methods and Properties in Structure 151](#_Toc95305305)

[C# - Conversion – Boxing & Unboxing, User Defined 152](#_Toc95305306)

[Boxing 152](#_Toc95305307)

[Unboxing 153](#_Toc95305308)

[User Defined Conversion 154](#_Toc95305309)

[C# - Generics 156](#_Toc95305310)

[Generic Class 156](#_Toc95305311)

[Features of Generics 158](#_Toc95305312)

[Advantages of Generics 159](#_Toc95305313)

[C# - Logging (Serilog) 160](#_Toc95305314)

[Minimum level 160](#_Toc95305315)

[Getting Started 161](#_Toc95305316)

[Let’s create a .NET Core Console application 161](#_Toc95305317)

[Add basic File Logging 161](#_Toc95305318)

[Add Rolling File Logging 162](#_Toc95305319)

[Another Example 163](#_Toc95305320)

[Serilog Configuration File 166](#_Toc95305321)

[File Sink 167](#_Toc95305322)

[Overriding per sink 168](#_Toc95305323)

[Enrichers 168](#_Toc95305324)

[Refs 169](#_Toc95305325)

[Unit Testing using xUnit 170](#_Toc95305326)

[Create the solution 170](#_Toc95305327)

[Commands to create the solution 171](#_Toc95305328)

[Create a test 172](#_Toc95305329)

[Add more tests 173](#_Toc95305330)

[Unit Test Code Coverage & Reporting 175](#_Toc95305331)

[System under test 175](#_Toc95305332)

[Create a class library 175](#_Toc95305333)

[Create test projects 176](#_Toc95305334)

[Create a solution 177](#_Toc95305335)

[Code coverage tooling 178](#_Toc95305336)

[Integrate with .NET test 178](#_Toc95305337)

[Generate reports 179](#_Toc95305338)

[Mocking using Moq (Dependency Injection) 180](#_Toc95305339)

[ADO.NET 183](#_Toc95305340)

[ADO.NET Architecture 183](#_Toc95305341)

[.NET Framework Data Providers 183](#_Toc95305342)

[The DataSet 183](#_Toc95305343)

[Choosing a DataReader or a DataSet 184](#_Toc95305344)

[LINQ to DataSet 185](#_Toc95305345)

[LINQ to SQL 185](#_Toc95305346)

[ADO.NET Entity Framework 185](#_Toc95305347)

[WCF Data Services 185](#_Toc95305348)

[XML and ADO.NET 185](#_Toc95305349)

[LINQ 185](#_Toc95305350)

[LINQ to DataSet 186](#_Toc95305351)

[LINQ to SQL 186](#_Toc95305352)

[LINQ to Entities 187](#_Toc95305353)

[.NET Framework Data Providers 187](#_Toc95305354)

[Core Objects of .NET Framework Data Providers 188](#_Toc95305355)

[.NET Framework Data Provider for SQL Server (SqlClient) 188](#_Toc95305356)

[ADO.NET SqlClient Data Provider Examples 189](#_Toc95305357)

[Retrieve Data Using a DataReader 189](#_Toc95305358)

[Closing the DataReader 190](#_Toc95305359)

[Executing a Command 191](#_Toc95305360)

[Getting a Single Value from a Table 193](#_Toc95305361)

[Updating Data 193](#_Toc95305362)

[Example 194](#_Toc95305363)

[DataSet and DataAdapter 195](#_Toc95305364)

[Example 196](#_Toc95305365)

[Multiple Result Sets 196](#_Toc95305366)

[Transactions and Concurrency 197](#_Toc95305367)

[Local Transactions 198](#_Toc95305368)

[Stored Procedures 200](#_Toc95305369)

[Example 200](#_Toc95305370)

[REST and SOA 203](#_Toc95305371)

[Service Oriented Architecture 203](#_Toc95305372)

[Service-Oriented Architecture (SOA) Principles 203](#_Toc95305373)

[Service-Oriented Architecture (SOA) Terminology 204](#_Toc95305374)

[Service-Oriented Architecture (SOA) Advantages 205](#_Toc95305375)

[Disadvantages of Service-Oriented Architecture (SOA) 205](#_Toc95305376)

[Asynchronous Programming (async, await, Task, Thread) 205](#_Toc95305377)

[Async improves responsiveness 205](#_Toc95305378)

[Async methods are easy to write 206](#_Toc95305379)

[What happens in an async method 207](#_Toc95305380)

[Another Simpler Example 209](#_Toc95305381)

[Async File I/O 212](#_Toc95305382)

[Reflection in .NET 215](#_Toc95305383)

[What is .NET Reflection? 215](#_Toc95305384)

[Roadmap 215](#_Toc95305385)

[System.Reflection Namespace 216](#_Toc95305386)

[The System.Type Class 218](#_Toc95305387)

[Using System.Object.GetType() 219](#_Toc95305388)

[Using System.Type.GetType() 220](#_Toc95305389)

[Using typeof() C# operator 221](#_Toc95305390)

[Type Properties 221](#_Toc95305391)

[Type Methods 224](#_Toc95305392)

[Reflecting on Methods 224](#_Toc95305393)

[Reflecting on Fields and Properties 226](#_Toc95305394)

[Reflecting on Implemented Interfaces 227](#_Toc95305395)

[Reflecting on Method Parameters and Return Values 228](#_Toc95305396)

[Reflecting on Constructor 229](#_Toc95305397)

[Assembly Class 229](#_Toc95305398)

[Dynamically loading an Assembly 230](#_Toc95305399)

[Late Binding 231](#_Toc95305400)

[Design Pattern – Factory Method 233](#_Toc95305401)

[Factory design pattern 233](#_Toc95305402)

[Who is the Gang of Four? 233](#_Toc95305403)

[Where to Use It? 233](#_Toc95305404)

[Factory design pattern in real-world example 234](#_Toc95305405)

[Who is what? 234](#_Toc95305406)

[REST – REpresentational State Transfer 241](#_Toc95305407)

[Guiding Principles of REST 241](#_Toc95305408)

[Uniform Interface 241](#_Toc95305409)

[Client-server 241](#_Toc95305410)

[Stateless 242](#_Toc95305411)

[Cacheable 242](#_Toc95305412)

[Layered system 242](#_Toc95305413)

[Code on demand (optional) 242](#_Toc95305414)

[What is a Resource? 242](#_Toc95305415)

[Resource Identifiers 243](#_Toc95305416)

[Hypermedia 243](#_Toc95305417)

[Self-descriptive 243](#_Toc95305418)

[Resource Methods 243](#_Toc95305419)

[REST and HTTP are not same 244](#_Toc95305420)

[Summary 244](#_Toc95305421)

[REST – HATEOAS 245](#_Toc95305422)

[How to Implement HATEOAS 246](#_Toc95305423)

[ASP.NET Core 247](#_Toc95305424)

[Why choose ASP.NET Core? 247](#_Toc95305425)

[Build web APIs and web UI using ASP.NET Core MVC 247](#_Toc95305426)

[Client-side development 248](#_Toc95305427)

[Framework selection 248](#_Toc95305428)

[Demo: Create an ASP.NET Core Web App – Visual Studio 248](#_Toc95305429)

[Demo: Create an ASP.NET Core Web App – Visual Studio Code 248](#_Toc95305430)

[Create a web app project 248](#_Toc95305431)

[Trust the development certificate 248](#_Toc95305432)

[Run the app 249](#_Toc95305433)

[ASP.NET Core Fundamentals 249](#_Toc95305434)

[Program.cs 249](#_Toc95305435)

[Dependency injection (services) 250](#_Toc95305436)

[Middleware 251](#_Toc95305437)

[Host 251](#_Toc95305438)

[Non-web scenarios 252](#_Toc95305439)

[Configuration 253](#_Toc95305440)

[Environments 253](#_Toc95305441)

[Logging 254](#_Toc95305442)

[Routing 255](#_Toc95305443)

[Error handling 255](#_Toc95305444)

[Make HTTP requests 256](#_Toc95305445)

[Content root 256](#_Toc95305446)

[Web root 256](#_Toc95305447)

[Dependency Injection 257](#_Toc95305448)

[ASP.NET Core – Middleware 257](#_Toc95305449)

[Configure Middleware 258](#_Toc95305450)

[Understand Run Method 259](#_Toc95305451)

[Configure Multiple Middleware (Demo2) 260](#_Toc95305452)

[Add Built-in Middleware Via NuGet 261](#_Toc95305453)

[Branch the middleware pipeline (Demo4) 261](#_Toc95305454)

[MVC Pattern 263](#_Toc95305455)

[Model Responsibilities 263](#_Toc95305456)

[View Responsibilities 263](#_Toc95305457)

[Controller Responsibilities 264](#_Toc95305458)

[ASP.NET Controllers 265](#_Toc95305459)

[Defining Actions 265](#_Toc95305460)

[Controller Helper Methods 266](#_Toc95305461)

[1. Methods resulting in an empty response body 266](#_Toc95305462)

[2. Methods resulting in a non-empty response body with a predefined content type 266](#_Toc95305463)

[3. Methods resulting in a non-empty response body formatted in a content type negotiated with the client 267](#_Toc95305464)

[Routing to controller actions in ASP.NET Core 267](#_Toc95305465)

[Set up conventional route 267](#_Toc95305466)

[Multiple conventional routes 269](#_Toc95305467)

[ASP.NET - Model Binding 270](#_Toc95305468)

[ASP.NET – Filters 271](#_Toc95305469)

[How filters work 271](#_Toc95305470)

[Filter types 271](#_Toc95305471)

[Action Filter Attribute 272](#_Toc95305472)

[The Scope of Action Filters 274](#_Toc95305473)

[Order of Execution 274](#_Toc95305474)

[Changing the Order 274](#_Toc95305475)

[Adding Validations 275](#_Toc95305476)

[Service Filter Attribute 277](#_Toc95305477)

[Attribute routing for REST APIs 279](#_Toc95305478)

# .NET Stack

## The history behind .NET: from the .NET Framework first release to .NET 6

Microsoft started working on the .NET framework in the late 90s. The idea was to create a platform based on so-called managed code, code that can be executed under a runtime environment. This was needed to improve the development experience and relieve engineers from handling security operations, active memory management, and other low-level efforts that C/C++ developers had to bother with.

## The .NET Framework era

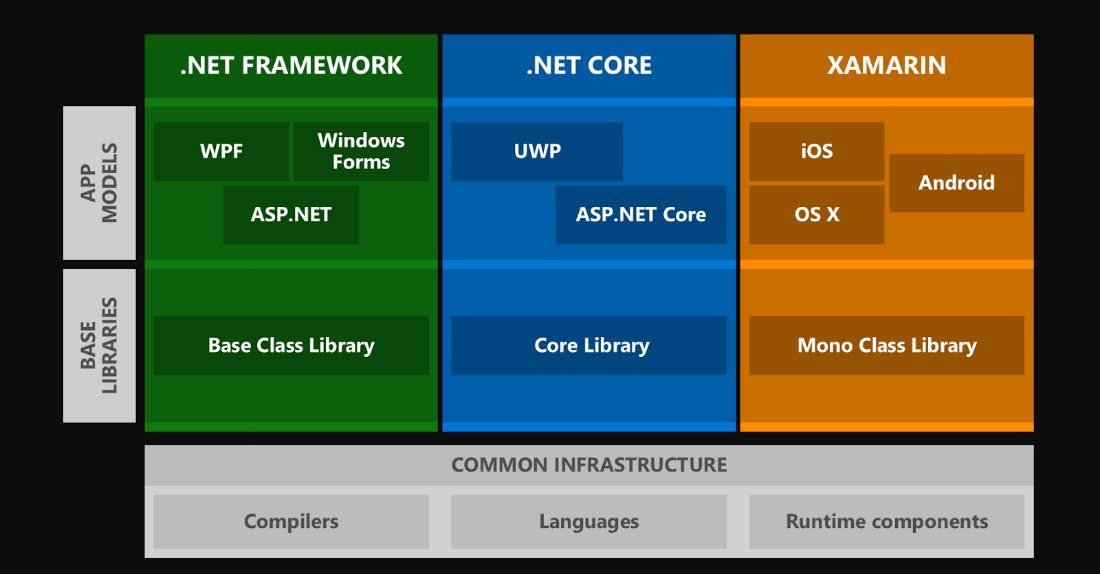
The first release of .NET Framework in 2002 introduced C#, a language for writing managed code that had a design similar to C++. The framework itself was aimed at Windows-based computers and servers. It had WinForms, a GUI library for desktop applications; ASP.NET, a framework for Web; and ADO.NET for data access. All these elements were driven by Common Language Runtime (CLR) to compile and execute managed code.

To unite various functions, .NET offered a framework class library (FCL) that included the base class library (BCL), network library, a numerics library, and others.

Since that time, the framework has undergone multiple iterations spanning runtime updates, new desktop graphical systems (WPF), APIs for service-oriented applications (WCF), and more.

## The .NET CORE era

In 2014, Microsoft announced a dramatic shift in the way .NET exists by presenting .NET Core, a new cross-platform, cloud-friendly, and open-source version of the framework. .NET Core made it to a release in 2016, becoming the main technology to consider for new .NET projects. Gradually, Microsoft started porting existing services to work with Core. Some that didn’t receive official ports, like Windows Communication Foundation (WCF), were substituted by alternatives sourced from the community.

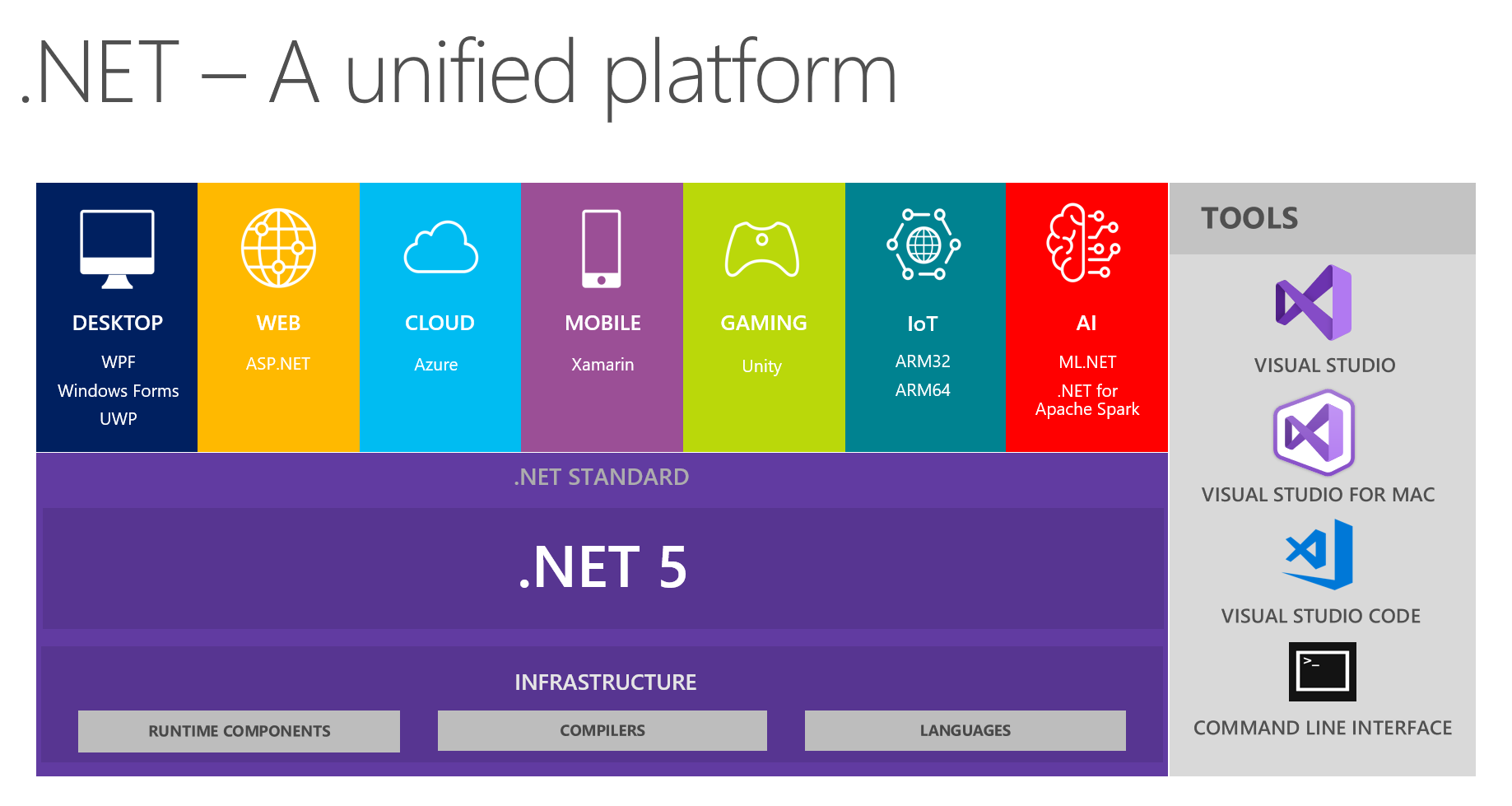


Also, in 2016, Microsoft acquired Xamarin, previously a proprietary technology for cross-platform mobile development, making it open source as well.

Microsoft continued moving towards “transparency between the product team and the community,” and open-sourced Windows Presentation Foundation (WPF), Windows Forms, and WinUI frameworks in December 2018.

## The .NET 5 and .NET 6 era

In May 2019, the company announced the big release that would tie the ecosystem together: All .NET elements were supposed to be bundled in the .NET 5 development platform. While changes were made to the schedule because of COVID-19, the .NET 5 unified development platform was finally introduced in November 2020. The successor to .NET Core 3.1 and .NET Framework 4.8, .NET 5 puts order into the fragmentation of the .NET world and provides a lot of features to build applications on Windows, Linux, macOS, iOS, watchOS, Android, tvOS, or using WebAssembly. The platform comes with new APIs, language features, and runtime capabilities. Also, .NET 5 includes ASP.NET Core, Xamarin, Entity Framework Core, WPF, WinForms, and ML.NET.



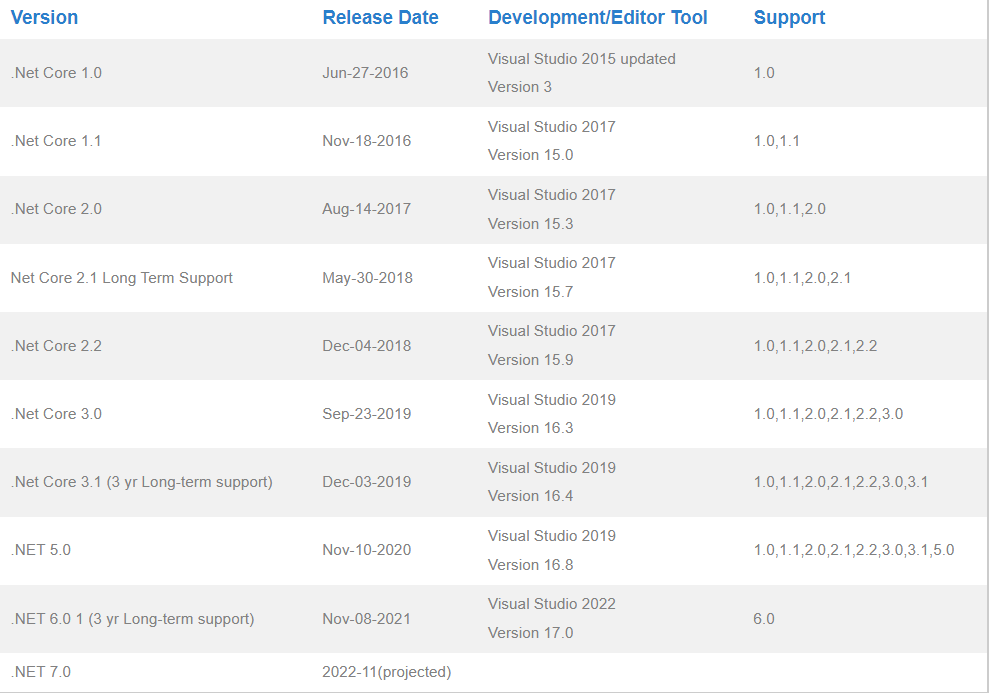
While .NET 5 set the unification foundations, the newest .NET 6 version delivered the final parts of it in November 2021, with Visual Studio 2022 released the same day. This is a unified platform for building projects across cloud, browser, IoT, mobile, and desktop environments, enabling all to use the same .NET libraries, SDK, and runtime.



One of the most prominent .NET 6 features is .NET MAUI (Multi-platform App UI) that acts as the cross-platform framework for developing native desktop and mobile apps with C# and XAML.

Apart from being the final step of the unification, .NET 6 can boast of:

* Better performance with decreased project execution time, latency time, and memory use.
* APIs for HTTP/3, JSON processing, mathematics, and direct memory manipulation.
* Improved security with support for OpenSSL 3 and ChaCha20Poly1305 encryption scheme.
* Stable version and long-term support for 3 years.



Besides all that, it’s the very first release that has native support for Apple Silicon (mac OS Arm64) and improved support for Windows Arm64.

# What is .NET development platform

.NET comes in four flavors: .NET Framework, .NET Core, Xamarin, and Universal Windows Platform (UWP).

These implementations combined are called the .NET development platform. Each of them contains frameworks and libraries to build various applications.

## .NET framework

The .NET Framework released back in 2002 is the first and oldest implementation of the platform. It includes three main application models – WPF, Windows Forms, ASP.NET Forms – and Base Class Library.

Windows Presentation Foundation (WPF) is a UI framework used for creating graphical interfaces primarily for desktop client applications on Windows OS. WPF uses the capabilities of Extensible Application Markup Language (XAML).

Windows Forms is a GUI class library within .NET Framework. Windows Forms are used to develop desktop applications with rich graphics that are easy to update and deploy.

ASP.NET. While the previous two components are designed for desktop engineering ASP.NET is used to develop dynamic websites and web applications. There is the Common Language Runtime (CLR) in its core that gives developers the opportunity to write ASP.NET code using different .NET languages that we discuss below.

Base Class Library (BCL) provides the most common functionality like classes in namespaces and is the core of the Framework Class Library (FCL), a set of reusable interfaces, classes, and value types that are closely integrated with the Common Language Runtime (CLR). The combination of FCL and CLR constitutes the .NET Framework. The base class library also includes ADO.NET, data access technology used by developers to access databases.

As .NET Framework supports only Windows-based devices, there was a need for a cross-platform package.

## .NET Core

.NET Core was released in 2016. It’s a cross-platform re-build of .NET Framework. Unlike the old version, engineers can now use the product on Linux and macOS and create applications that aren’t necessarily tied to the Windows family. The new system aims at conquering the cloud space as some providers like Digital Ocean are Linux-driven. Not only is .NET Core cross-platform but its different versions can also be installed side by side on the same device. .NET Core includes ASP.NET Core and Universal Windows Platform (UWP).

ASP.NET Core is a rebuild of ASP.NET that happened to be a more modular framework than its predecessor. ASP.NET Core allows you to build the mobile backend, web apps, and services. It’s also cross-platform and runs on OS X, Windows, and Linux.

In .NET 6, ASP.NET Core counts many new features including Hot Reload that enables the application of changes to C#, Visual Basic, and CSS source files without having to restart or rebuild the app (the app is running while you are updating the code).

## Xamarin

The third implementation is called Xamarin and is used for mobile applications and Mac products. Originally, Xamarin was designed independently from Microsoft and was a proprietary product. Then Microsoft acquired it in 2016 making it a fully open-source branch of the .NET platform. Xamarin uses the Mono runtime and a version of the .NET Framework adjusted to work with APIs for iOS, Android, and Xamarin.Mac.

All runtimes use a common infrastructure that makes the entire ecosystem work. It provides runtime components, languages, and compilers.

## Universal Windows Platform (UWP)

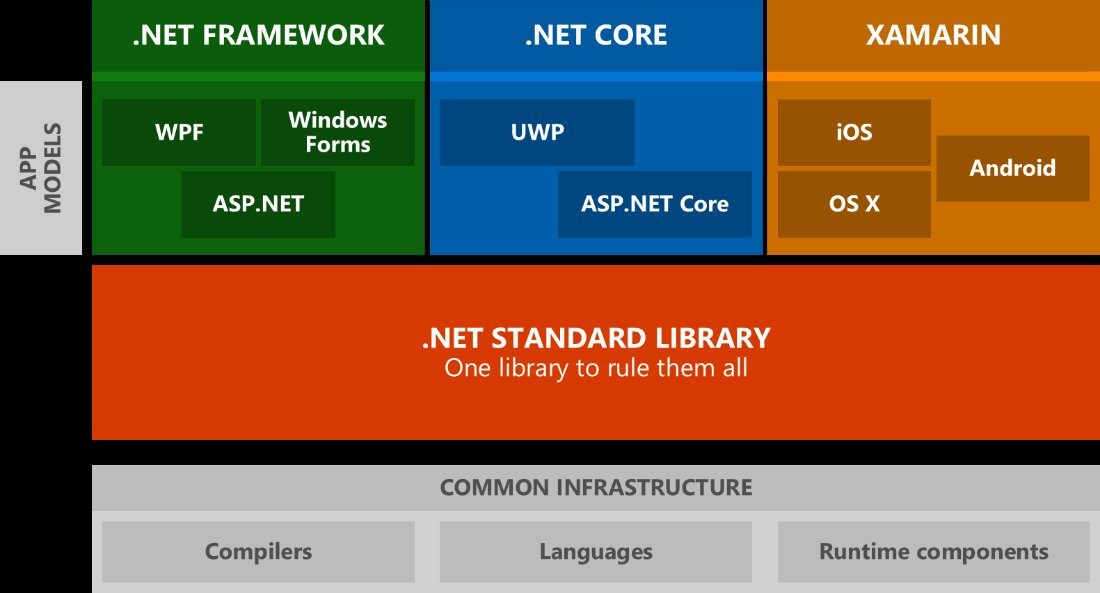
UWP provides a common type system, APIs, and application model for all devices running on Windows 10. So, UWP enables the development of universal apps for PC, tablet, Xbox, Surface Hub, HoloLens, or Internet of Things (IoT) devices.

UWP app developers get access to the Microsoft store that charges only 15 percent for non-gaming subscription-based apps, unlike Google Play Store and App Store. Other services include an execution environment (AppContainer) and Extension SDKs to call specialized APIs for different devices.

Unfortunately, .NET 5 and .NET 6 will not be coming to UWP project types and there will not be an update on UWP in this regard.

## .NET Standard

In 2016 Microsoft also introduced .NET Standard, a library that combines APIs from .NET Framework, .NET Core, and Xamarin allowing engineers to use a single base-class library rather than mastering three different ones related to each .NET implementation. This step unified the ecosystem and brought a higher consistency to reusing components across different platforms.



.NET development platform is best served with Visual Studio IDE used for building, debugging, and publishing applications across all platforms and devices.

## Common language runtime (CLR)

Common Language Runtime (CLR) is the heart of .NET, an application virtual machine that manages memory, implements code access security, verifies code safety, and provides execution of threads and code. CLR is what makes the .NET code a managed one.

As we mentioned, the idea of CLR is to make the developer’s life easier. Besides, it allows engineers to design systems with multiple languages, as CLR enables them to communicate and integrate their behaviors. The runtime checks the needed versions of applied services to ensure that all dependencies are intact and the code works as intended.

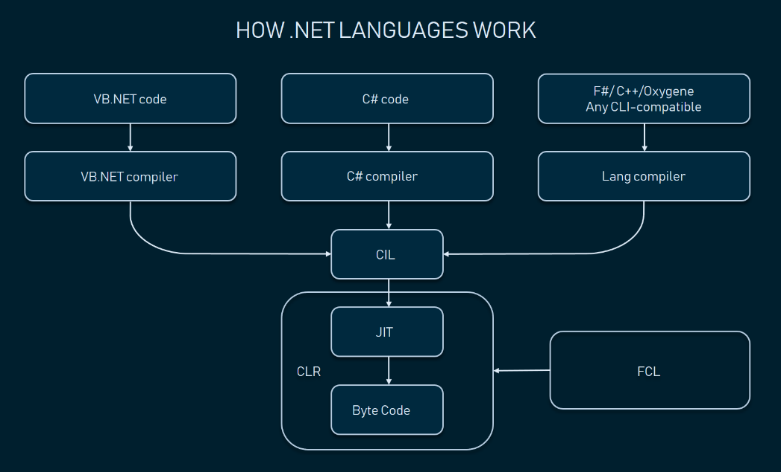
In .NET Core, an open-source CoreCLR is used. While nearly identical to CLR in .NET Framework, CoreCLR is adjusted to the .NET Core cross-platform makeup.

## Supported languages

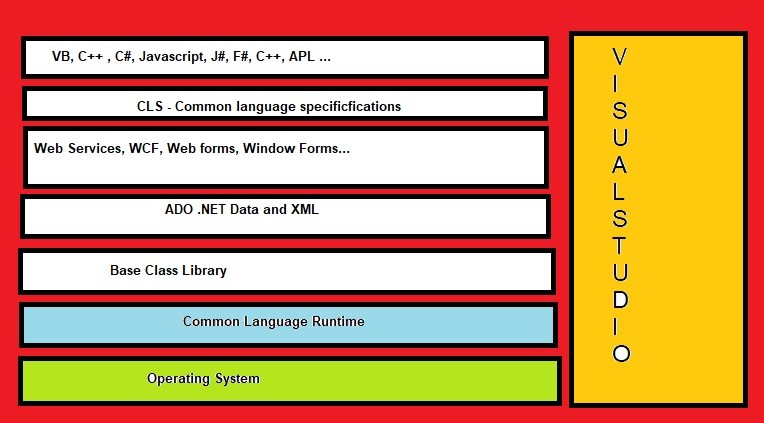
The languages that you can use with .NET can be ultimately divided into two major groups: 1) the main officially supported languages by Microsoft, and 2) the rest of the languages that are CLI-compliant.

Main languages. Most of the .NET development happens with C#, F#, and Visual Basic. All three saw great improvements within the .NET 6 release. C# 10 and F# 6 were introduced, both aiming at simplifying code and making it more performant.

The rest of CLI-compliant languages. CLI means common language infrastructure. It’s a technical standard for high-level languages that can be compiled into a common intermediate language (CIL) and further compiled into a byte-code. Besides those three mentioned above, there are about 25 active CLI-compliant languages, including C++/CLI, IronPython, Oxygene, Phalanger, and more. There’s also a number of languages that are no longer used, like IronRuby.



## The Framework



**IL/MSIL/CIL**– Intermediate Language (IL) code is a CPU independent partially compiled code. It’s partially compiled because we do not know in what kind of environment .NET code will run and on runtime IL Code will compile to machine code using the environmental properties (CPU, OS, machine configuration, etc).

**ILDASM**– IL Disassembler. This is a tool provided by Visual Studio to view IL code. To run ILDASM, we have to select option “Visual Studio Command Prompt” from “Visual Studio Tools” and type ildasm. It will open the ildasm tool where we can open any *exe/dll.ildasm* tool read the assembly by reflection and it is showing us various properties, methods which our assembly has. Here, we can see IL code of any method/property by clicking on that.

**CLR**– Common Language Runtinmg. CLR is the heart of the .NET framework and it does 4 primary important things:

1. Garbage collection
2. CAS (Code Access Security)
3. CV (Code Verification)
4. IL to Native translation

**CTS** – Common Type System. CTS ensures that data types defined in two different languages get compiled to a common data type. This is useful because there may be situations when we want code in one language to be called in other language.

We can see a practical demonstration of CTS by creating the same application in C# and VB.NET and then compare the IL code of both applications. Here, the datatype of both IL code is same.

**CLS** – Common Language Specification. CLS is a subset of CTS. CLS is a set of rules or guidelines. When any programming language adheres to these set of rules, it can be consumed by any .NET language.

Diagram

Description automatically generated

**VES(Virtual Execution System):**VES known as CLR(Common Language Runtime), all.Net languages after compilation gets converted into CIL code. This CIL code can be carried and executed on any machine provided as operating system specific VES or CLR is available which converts CIL code into machine code according to the platform.

Timeline

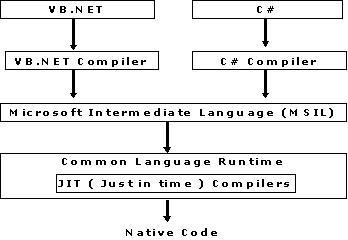
Description automatically generated

**JIT**– Just In Time. JIT compiles the IL code to Machine code just before execution and then saves this transaction in memory.

**JIT has three types:**

* **Normal: This complies only those methods that are called at runtime. These methods are compiled only first time when they are called, and then they are stored in memory cache. This memory cache is commonly called as JITTED. When the same methods are called again, the complied code from cache is used for execution.**
* **Ecno: This complies only those methods that are called at runtime and removes them from memory after execution.**
* **Pre: This complies entire MSIL code into native code in a single compilation cycle. This is done at the time of deployment of the application.**

**BCL** – Base Class Library (or **FCL** – Framework Class Library)



### Stack vs Heap

The stack is the memory set aside as scratch space for a thread of execution. When a function is called, a block is reserved on the top of the stack for local variables and some bookkeeping data. When that function returns, the block becomes unused and can be used the next time a function is called. The stack is always reserved in a LIFO (last in first out) order; the most recently reserved block is always the next block to be freed. This makes it really simple to keep track of the stack; freeing a block from the stack is nothing more than adjusting one pointer.

The heap is memory set aside for dynamic allocation. Unlike the stack, there's no enforced pattern to the allocation and deallocation of blocks from the heap; you can allocate a block at any time and free it at any time. This makes it much more complex to keep track of which parts of the heap are allocated or free at any given time; there are many custom heap allocators available to tune heap performance for different usage patterns.

Each thread gets a stack, while there's typically only one heap for the application (although it isn't uncommon to have multiple heaps for different types of allocation).

**Stack:**

* Stored in computer RAM just like the heap.
* Variables created on the stack will go out of scope and are automatically deallocated.
* Much faster to allocate in comparison to variables on the heap.
* Implemented with an actual stack data structure.
* Stores local data, return addresses, used for parameter passing.
* Can have a stack overflow when too much of the stack is used (mostly from infinite or too deep recursion, very large allocations).
* Data created on the stack can be used without pointers.
* You would use the stack if you know exactly how much data you need to allocate before compile time and it is not too big.
* Usually has a maximum size already determined when your program starts.

**Heap:**

* Stored in computer RAM just like the stack.
* In C++, variables on the heap must be destroyed manually and never fall out of scope. The data is freed with delete, delete[], or free.
* Slower to allocate in comparison to variables on the stack.
* Used on demand to allocate a block of data for use by the program.
* Can have fragmentation when there are a lot of allocations and deallocations.
* In C++ or C, data created on the heap will be pointed to by pointers and allocated with new or malloc respectively.
* Can have allocation failures if too big of a buffer is requested to be allocated.
* You would use the heap if you don't know exactly how much data you will need at run time or if you need to allocate a lot of data.
* Responsible for memory leaks.

|  |  |
| --- | --- |
| Enter image description here |  |

# C# - Input / Output

## C# Output

In order to output something in C#, we can use

System.Console.WriteLine() OR

System.Console.Write()

Here, System is a namespace, Console is a class within namespace System and WriteLine and Write are methods of class Console.

Let's look at a simple example that prints a string to output screen.

**Example 1**: Printing String using WriteLine()

using System;

namespace Sample

{

class Test

{

public static void Main(string[] args)

{

Console.WriteLine("C# is cool");

}

}

}

When we run the program, the output will be

C# is cool

### Difference between WriteLine() and Write() method

The main difference between WriteLine() and Write() is that the Write() method only prints the string provided to it, while the WriteLine() method prints the string and moves to the start of next line as well.

Let's take at a look at the example below to understand the difference between these methods.

**Example 2**: How to use WriteLine() and Write() method?

using System;

namespace Sample

{

class Test

{

public static void Main(string[] args)

{

Console.WriteLine("Prints on ");

Console.WriteLine("New line");

Console.Write("Prints on ");

Console.Write("Same line");

}

}

}

When we run the program, the output will be

Prints on

New line

Prints on Same line

### Printing Variables and Literals using WriteLine() and Write()

The WriteLine() and Write() method can be used to print variables and literals. Here's an example.

**Example 3**: Printing Variables and Literals

using System;

namespace Sample

{

class Test

{

public static void Main(string[] args)

{

int value = 10;

// Variable

Console.WriteLine(value);

// Literal

Console.WriteLine(50.05);

}

}

}

When we run the program, the output will be

10

50.05

### Combining (Concatenating) two strings using + operator and printing them

Strings can be combined/concatenated using the + operator while printing.

**Example 4**: Printing Concatenated String using + operator

using System;

namespace Sample

{

class Test

{

public static void Main(string[] args)

{

int val = 55;

Console.WriteLine("Hello " + "World");

Console.WriteLine("Value = " + val);

}

}

}

When we run the program, the output will be

Hello World

Value = 55

### Printing concatenated string using Formatted String [Better Alternative]

A better alternative for printing concatenated string is using formatted string. Formatted string allows programmer to use placeholders for variables. For example,

The following line,

Console.WriteLine("Value = " + val);

can be replaced by,

Console.WriteLine("Value = {0}", val);

{0} is the placeholder for variable val which will be replaced by value of val. Since only one variable is used so there is only one placeholder.

Multiple variables can be used in the formatted string. We will see that in the example below.

**Example 5**: Printing Concatenated string using String formatting

using System;

namespace Sample

{

class Test

{

public static void Main(string[] args)

{

int firstNumber = 5, secondNumber = 10, result;

result = firstNumber + secondNumber;

Console.WriteLine("{0} + {1} = {2}", firstNumber, secondNumber, result);

}

}

}

When we run the program, the output will be

5 + 10 = 15

Here, {0} is replaced by firstNumber, {1} is replaced by secondNumber and {2} is replaced by result. This approach of printing output is more readable and less error prone than using + operator.

## C# Input

In C#, the simplest method to get input from the user is by using the ReadLine() method of the Console class. However, Read() and ReadKey() are also available for getting input from the user. They are also included in Console class.

**Example 6**: Get String Input From User

using System;

namespace Sample

{

class Test

{

public static void Main(string[] args)

{

string testString;

Console.Write("Enter a string - ");

testString = Console.ReadLine();

Console.WriteLine("You entered '{0}'", testString);

}

}

}

When we run the program, the output will be:

Enter a string - Hello World

You entered 'Hello World'

### Difference between ReadLine(), Read() and ReadKey() method:

The difference between ReadLine(), Read() and ReadKey() method is:

* ReadLine(): The ReadLine() method reads the next line of input from the standard input stream. It returns the same string.
* Read(): The Read() method reads the next character from the standard input stream. It returns the ascii value of the character.
* ReadKey(): The ReadKey() method obtains the next key pressed by user. This method is usually used to hold the screen until user press a key.

**Example 7**: Difference between Read() and ReadKey() method

using System;

namespace Sample

{

class Test

{

public static void Main(string[] args)

{

int userInput;

Console.WriteLine("Press any key to continue...");

Console.ReadKey();

Console.WriteLine();

Console.Write("Input using Read() - ");

userInput = Console.Read();

Console.WriteLine("Ascii Value = {0}",userInput);

}

}

}

When we run the program, the output will be

Press any key to continue...

x

Input using Read() - Learning C#

Ascii Value = 76

From this example, it must be clear how ReadKey() and Read() method works. While using ReadKey(), as soon as the key is pressed, it is displayed on the screen.

When Read() is used, it takes a whole line but only returns the ASCII value of first character. Hence, 76 (ASCII value of L) is printed.

### Reading numeric values (integer and floating point types)

Reading a character or string is very simple in C#. All you need to do is call the corresponding methods as required.

But, reading numeric values can be slightly tricky in C#. We’ll still use the same ReadLine() method we used for getting string values. But since the ReadLine() method receives the input as string, it needs to be converted into integer or floating point type.

One simple approach for converting our input is using the methods of Convert class.

**Example 8**: Reading Numeric Values from User using Convert class

using System;

namespace UserInput

{

class MyClass

{

public static void Main(string[] args)

{

string userInput;

int intVal;

double doubleVal;

Console.Write("Enter integer value: ");

userInput = Console.ReadLine();

/\* Converts to integer type \*/

intVal = Convert.ToInt32(userInput);

Console.WriteLine("You entered {0}",intVal);

Console.Write("Enter double value: ");

userInput = Console.ReadLine();

/\* Converts to double type \*/

doubleVal = Convert.ToDouble(userInput);

Console.WriteLine("You entered {0}",doubleVal);

}

}

}

When we run the program, the output will be

Enter integer value: 101

You entered 101

Enter double value: 59.412

You entered 59.412

The ToInt32() and ToDouble() method of Convert class converts the string input to integer and double type respectively. Similarly we can convert the input to other types.

# C# - Data Types (Primitive)

C# is a strongly-typed language. It means we must declare the type of a variable that indicates the kind of values it is going to store, such as integer, float, decimal, text, etc.

The following declares and initialized variables of different data types.

**Example**: Variables of Different Data Types

string stringVar = "Hello World!!";

int intVar = 100;

float floatVar = 10.2f;

char charVar = 'A';

bool boolVar = true;

C# mainly categorized data types in two types: Value types and Reference types. Value types include simple types (such as int, float, bool, and char), enum types, struct types, and Nullable value types. Reference types include class types, interface types, delegate types, and array types.



## Predefined Data Types in C#

C# includes some predefined value types and reference types. The following table lists predefined data types:

| Type | Description | Range | Suffix |
| --- | --- | --- | --- |
| byte | 8-bit unsigned integer | 0 to 255 |  |
| sbyte | 8-bit signed integer | -128 to 127 |  |
| short | 16-bit signed integer | -32,768 to 32,767 |  |
| ushort | 16-bit unsigned integer | 0 to 65,535 |  |
| int | 32-bit signed integer | -2,147,483,648 to 2,147,483,647 |  |
| uint | 32-bit unsigned integer | 0 to 4,294,967,295 | u |
| long | 64-bit signed integer | -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807 | l |
| ulong | 64-bit unsigned integer | 0 to 18,446,744,073,709,551,615 | ul |
| float | 32-bit Single-precision floating point type | -3.402823e38 to 3.402823e38 | f |
| double | 64-bit double-precision floating point type | -1.79769313486232e308 to 1.79769313486232e308 | d |
| decimal | 128-bit decimal type for financial and monetary calculations | (+ or -)1.0 x 10e-28 to 7.9 x 10e28 | m |
| char | 16-bit single Unicode character | Any valid character, e.g. a,\*, \x0058 (hex), or\u0058 (Unicode) |  |
| bool | 8-bit logical true/false value | True or False |  |
| object | Base type of all other types. |  |  |
| string | A sequence of Unicode characters |  |  |
| DateTime | Represents date and time | 0:00:00am 1/1/01 to 11:59:59pm 12/31/9999 |  |

As you can see in the above table that each data type (except string and object) includes value range. The compiler will give an error if the value goes out of datatype's permitted range. For example, int data type's range is -2,147,483,648 to 2,147,483,647. So if you assign a value which is not in this range, then the compiler would give an error.

**Example**: Compile time error

// compile time error: Cannot implicitly convert type 'long' to 'int'.

int i = 21474836470;

The value of unsigned integers, long, float, double, and decimal type must be suffix by u,l,f,d, and m, respectively.

Example: Value Suffix

uint ui = 100u;

float fl = 10.2f;

long l = 45755452222222l;

ulong ul = 45755452222222ul;

double d = 11452222.555d;

decimal mon = 1000.15m;

## Alias vs .NET Type

The predefined data types are alias to their .NET type (CLR class) name. The following table lists alias for predefined data types and related .NET class name.

| Alias | .NET Type | Type |
| --- | --- | --- |
| byte | System.Byte | struct |
| sbyte | System.SByte | struct |
| int | System.Int32 | struct |
| uint | System.UInt32 | struct |
| short | System.Int16 | struct |
| ushort | System.UInt16 | struct |
| long | System.Int64 | struct |
| ulong | System.UInt64 | struct |
| float | System.Single | struct |
| double | System.Double | struct |
| char | System.Char | struct |
| bool | System.Boolean | struct |
| object | System.Object | Class |
| string | System.String | Class |
| decimal | System.Decimal | struct |
| DateTime | System.DateTime | struct |

It means that whether you define a variable of int or Int32, both are the same.

int i = 345;

Int32 i = 345;// same as above

## Default Values

Every data type has a default value. Numeric type is 0, boolean has false, and char has '\0' as default value. Use the default(typename) to assign a default value of the data type or C# 7.1 onward, use default literal.

int i = default(int); // 0

float f = default(float);// 0

decimal d = default(decimal);// 0

bool b = default(bool);// false

char c = default(char);// '\0'

// C# 7.1 onwards

int i = default; // 0

float f = default;// 0

decimal d = default;// 0

bool b = default;// false

char c = default;// '\0'

## Conversions

The values of certain data types are automatically converted to different data types in C#. This is called an implicit conversion.

**Example**: Implicit Conversion

int i = 345;

float f = i;

Console.WriteLine(f); //output: 345

In the above example, the value of an integer variable i is assigned to the variable of float type f because this conversion operation is predefined in C#.

The following is an implicit data type conversion table.

| Implicit Conversion From | To |
| --- | --- |
| sbyte | short, int, long, float, double, decimal |
| byte | short, ushort, int, uint, long, ulong, float, double, decimal |
| short | int, long, float, double, or decimal |
| ushort | int, uint, long, ulong, float, double, or decimal |
| int | long, float, double, or decimal. |
| uint | long, ulong, float, double, or decimal |
| long | float, double, or decimal |
| ulong | float, double, or decimal |
| char | ushort, int, uint, long, ulong, float, double, or decimal |
| float | Double |

Conversions from int, uint, long, or ulong to float and from long or ulong to double may cause a loss of precision. No data type implicitly converted to the char type.

However, not all data types are implicitly converted to other data types. For example, int type cannot be converted to uint implicitly. It must be specified explicitly, as shown below.

**Example**: Explicit Conversion

public static void Main()

{

int i = 100;

uint u = (uint) i;

Console.Write(i);

}

## Enums

In C#, an enum (or enumeration type) is used to assign constant names to a group of numeric integer values. It makes constant values more readable, for example, WeekDays.Monday is more readable then number 0 when referring to the day in a week.

An enum is defined using the enum keyword, directly inside a namespace, class, or structure. All the constant names can be declared inside the curly brackets and separated by a comma. The following defines an enum for the weekdays.

**Example**: Define an Enum

enum WeekDays

{

Monday,

Tuesday,

Wednesday,

Thursday,

Friday,

Saturday,

Sunday

}

Above, the WeekDays enum declares members in each line separated by a comma.

## Enum Values

If values are not assigned to enum members, then the compiler will assign integer values to each member starting with zero by default. The first member of an enum will be 0, and the value of each successive enum member is increased by 1.

**Example**: Default Enum Values

enum WeekDays

{

Monday, // 0

Tuesday, // 1

Wednesday, // 2

Thursday, // 3

Friday, // 4

Saturday, // 5

Sunday // 6

}

You can assign different values to enum member. A change in the default value of an enum member will automatically assign incremental values to the other members sequentially.

**Example**: Assign Values to Enum Members

enum Categories

{

Electronics, // 0

Food, // 1

Automotive = 6, // 6

Arts, // 7

BeautyCare, // 8

Fashion // 9

}

You can even assign different values to each member.

**Example**: Assign Values to Enum Members

enum Categories

{

Electronics = 1,

Food = 5,

Automotive = 6,

Arts = 10,

BeautyCare = 11,

Fashion = 15,

WomanFashion = 15

}

The enum can be of any numeric data type such as byte, sbyte, short, ushort, int, uint, long, or ulong. However, an enum cannot be a string type.

Specify the type after enum name as : type. The following defines the byte enum.

**Example**: byte Enum

enum Categories: byte

{

Electronics = 1,

Food = 5,

Automotive = 6,

Arts = 10,

BeautyCare = 11,

Fashion = 15

}

## Access an Enum

An enum can be accessed using the dot syntax: enum.member

**Example**: Access Enum

enum WeekDays

{

Monday,

Tuesday,

Wednesday,

Thursday,

Friday,

Saturday,

Sunday

}

Console.WriteLine(WeekDays.Monday); // Monday

Console.WriteLine(WeekDays.Tuesday); // Tuesday

Console.WriteLine(WeekDays.Wednesday); // Wednesday

Console.WriteLine(WeekDays.Thursday); // Thursday

Console.WriteLine(WeekDays.Friday); // Friday

Console.WriteLine(WeekDays.Saturday); // Saturday

Console.WriteLine(WeekDays.Sunday); // Sunday

## Conversion

Explicit casting is required to convert from an enum type to its underlying integral type.

**Example**: Enum Conversion

enum WeekDays

{

Monday,

Tuesday,

Wednesday,

Thursday,

Friday,

Saturday,

Sunday

}

Console.WriteLine(WeekDays.Friday); //output: Friday

int day = (int) WeekDays.Friday; // enum to int conversion

Console.WriteLine(day); //output: 4

var wd = (WeekDays) 5; // int to enum conversion

Console.WriteLine(wd);//output: Saturday

# C# - Arrays

A variable is used to store a literal value, whereas an array is used to store multiple literal values.

An array is the data structure that stores a fixed number of literal values (elements) of the same data type. Array elements are stored contiguously in the memory.

In C#, an array can be of three types: single-dimensional, multidimensional, and jagged array. Here you will learn about the single-dimensional array.

The following figure illustrates an array representation.

[Table

Description automatically generated](https://www.tutorialsteacher.com/Content/images/csharp/array.png)

## Array Declaration and Initialization

An array can be declared using by specifying the type of its elements with square brackets.

Example: Array Declaration

int[] evenNums; // integer array

string[] cities; // string array

The following declares and adds values into an array in a single statement.

Example: Array Declaration & Initialization

int[] evenNums = new int[5]{ 2, 4, 6, 8, 10 };

string[] cities = new string[3]{ "Mumbai", "London", "New York" };

Above, evenNums array can store up to five integers. The number 5 in the square brackets new int[5] specifies the size of an array. In the same way, the size of cities array is three. Array elements are added in a comma-separated list inside curly braces { }.

Arrays type variables can be declared using [var](https://www.tutorialsteacher.com/csharp/csharp-var-implicit-typed-local-variable) without square brackets.

Example: Array Declaration using var

var evenNums = new int[]{ 2, 4, 6, 8, 10};

var cities = new string[]{ "Mumbai", "London", "New York" };

If you are adding array elements at the time of declaration, then size is optional. The compiler will infer its size based on the number of elements inside curly braces, as shown below.

Example: Short Syntax of Array Declaration

int[] evenNums = { 2, 4, 6, 8, 10};

string[] cities = { "Mumbai", "London", "New York" }

The following example demonstrate invalid array declarations.

Example: Invalid Array Creation

//must specify the size

int[] evenNums = new int[];

//number of elements must be equal to the specified size

int[] evenNums = new int[5] { 2, 4 };

//cannot use var with array initializer

var evenNums = { 2, 4, 6, 8, 10};

It is not necessary to declare and initialize an array in a single statement. You can first declare an array then initialize it later on using the new operator.

Example: Late Initialization

int[] evenNums;

evenNums = new int[5];

// or

evenNums = new int[]{ 2, 4, 6, 8, 10 };

## Accessing Array Elements

Array elements can be accessed using an index. An index is a number associated with each array element, starting with index 0 and ending with array size - 1.

The following example add/update and retrieve array elements using indexes.

Example: Access Array Elements using Indexes

int[] evenNums = new int[5];

evenNums[0] = 2;

evenNums[1] = 4;

//evenNums[6] = 12; //Throws run-time exception IndexOutOfRange

Console.WriteLine(evenNums[0]); //prints 2

Console.WriteLine(evenNums[1]); //prints 4

Note that trying to add more elements than its specified size will result in IndexOutOfRangeException.

## Accessing Array using for Loop

Use the for loop to access array elements. Use the length property of an array in conditional expression of the for loop.

Example: Accessing Array Elements using for Loop

int[] evenNums = { 2, 4, 6, 8, 10 };

for(int i = 0; i < evenNums.Length; i++)

Console.WriteLine(evenNums[i]);

for(int i = 0; i < evenNums.Length; i++)

evenNums[i] = evenNums[i] + 10; // update the value of each element by 10

## Accessing Array using foreach Loop

Use foreach loop to read values of an array elements without using index.

Example: Accessing Array using foreach Loop

int[] evenNums = { 2, 4, 6, 8, 10};

string[] cities = { "Mumbai", "London", "New York" };

foreach(var item in evenNums)

Console.WriteLine(item);

foreach(var city in cities)

Console.WriteLine(city);

## LINQ Methods

All the arrays in C# are derived from an abstract base class System.Array.

The Array class implements the IEnumerable interface, so you can LINQ extension methods such as Max(), Min(), Sum(), reverse(), etc.

Example: LINQ Methods

int[] nums = new int[5]{ 10, 15, 16, 8, 6 };

nums.Max(); // returns 16

nums.Min(); // returns 6

nums.Sum(); // returns 55

nums.Average(); // returns 55

The System.ArraySystem.Array class also includes methods for creating, manipulating, searching, and sorting arrays.

Example: Array Methods

int[] nums = new int[5]{ 10, 15, 16, 8, 6 };

Array.Sort(nums); // sorts array

Array.Reverse(nums); // sorts array in descending order

Array.ForEach(nums, n => Console.WriteLine(n)); // iterates array

Array.BinarySearch(nums, 5);// binary search

## Passing Array as Argument

An array can be passed as an argument to a method parameter. Arrays are reference types, so the method can change the value of the array elements.

Example: Passing Array as Argument

public static void Main(){

int[] nums = { 1, 2, 3, 4, 5 };

UpdateArray(nums);

foreach(var item in nums)

Console.WriteLine(item);

}

public static void UpdateArray(int[] arr)

{

for(int i = 0; i < arr.Length; i++)

arr[i] = arr[i] + 10;

}

i = 0; i < arr.Length; i++)

arr[i] = arr[i] + 10;

}

# C# - Multidimensional Arrays

C# supports multidimensional arrays up to 32 dimensions. The multidimensional array can be declared by adding commas in the square brackets. For example, [,] declares two-dimensional array, [, ,] declares three-dimensional array, [, , ,] declares four-dimensional array, and so on. So, in a multidimensional array, no of commas = No of Dimensions - 1.

The following declares multidimensional arrays.

Example: Multidimensional Arrays

int[,] arr2d; // two-dimensional array

int[, ,] arr3d; // three-dimensional array

int[, , ,] arr4d ; // four-dimensional array

int[, , , ,] arr5d; // five-dimensional array

Let's understand the two-dimensional array. The following initializes the two-dimensional array.

Example: two-dimensional Array

int[,] arr2d = new int[3,2]{

{1, 2},

{3, 4},

{5, 6}

};

// or

int[,] arr2d = {

{1, 2},

{3, 4},

{5, 6}

};

[,] arr2d = {

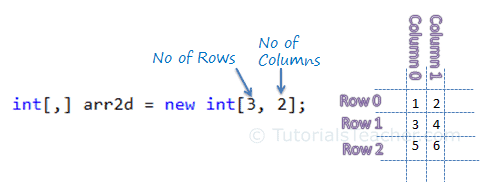
{1, 2},

{3, 4},

{5, 6}

};

In the above example of a two-dimensional array, [3, 2] defines the no of rows and columns. The first rank denotes the no of rows, and the second rank defines no of columns. The following figure illustrates the two-dimensional array divided into rows and columns.

[](https://www.tutorialsteacher.com/Content/images/csharp/twodimensional-array.PNG)

The following access values of the two-dimensional array.

Example: Access two-dimensional Array

int[,] arr2d = new int[3,2]{

{1, 2},

{3, 4},

{5, 6}

};

arr2d[0, 0]; //returns 1

arr2d[0, 1]; //returns 2

arr2d[1, 0]; //returns 3

arr2d[1, 1]; //returns 4

arr2d[2, 0]; //returns 5

arr2d[2, 1]; //returns 6

//arr2d[3, 0]; //throws run-time error as there is no 4th row

In the above example, the value of a two-dimensional array can be accessed by index no of row and column as [row index, column index]. So, [0, 0] returns the value of the first row and first column and [1, 1] returns the value from the second row and second column.

Now, let's understand the three-dimensional array. The following declares and initializes three-dimensional arrays.

Example: Three-dimensional Array

int[, ,] arr3d1 = new int[1, 2, 2]{

{ { 1, 2}, { 3, 4} }

};

int[, ,] arr3d2 = new int[2, 2, 2]{

{ {1, 2}, {3, 4} },

{ {5, 6}, {7, 8} }

};

int[, ,] arr3d3 = new int[2, 2, 3]{

{ { 1, 2, 3}, {4, 5, 6} },

{ { 7, 8, 9}, {10, 11, 12} }

};

arr3d2[0, 0, 0]; // returns 1

arr3d2[0, 0, 1]; // returns 2

arr3d2[0, 1, 0]; // returns 3

arr3d2[0, 1, 1]; // returns 4

arr3d2[1, 0, 0]; // returns 5

arr3d2[1, 0, 1]; // returns 6

arr3d2[1, 1, 0]; // returns 7

arr3d2[1, 1, 1]; // returns 8

As you can see in the above example, [1, 2, 2] of arr3d1 specifies that it will contain one row of two-dimensional array [2, 2]. arr3d2 specifies dimensions [2, 2, 2], which indicates that it includes two rows of two-dimensional array of [2, 2]. Thus, the first rank indicates the number of rows of inner two-dimensional arrays.

Now, consider the following four-dimensional array.

Example: Four-dimensional Array

int[,,,] arr4d1 = new int[1, 1, 2, 2]{

{

{ { 1, 2}, { 3, 4} }

}

};

arr4d1[0, 0, 0, 0]; // returns 1

arr4d1[0, 0, 0, 1]; // returns 2

arr4d1[0, 0, 1, 0]; // returns 3

arr4d1[0, 0, 1, 1]; // returns 4

int[,,,] arr4d2 = new int[1, 2, 2, 2]{

{

{ {1, 2}, {3, 4} },

{ {5, 6}, {7, 8} }

}

};

arr4d2[0, 0, 0, 0]; // returns 1

arr4d2[0, 0, 0, 1]; // returns 2

arr4d2[0, 0, 1, 0]; // returns 3

arr4d2[0, 0, 1, 1]; // returns 4

arr4d2[0, 1, 0, 0]; // returns 5

arr4d2[0, 1, 0, 1]; // returns 6

arr4d2[0, 1, 1, 0]; // returns 7

arr4d2[0, 1, 1, 1]; // returns 8

In the above example, the four-dimensional array arr4d1 specifies [1, 1, 2, 2], which indicates that it includes one row of the three-dimensional array.

In the same way, you can declare and initialize five-dimensional, six-dimensional array, and up to 32-dimensional arrays in C#.

# C# - Jagged Arrays: An Array of Array

A jagged array is an array of array. Jagged arrays store arrays instead of literal values.

A jagged array is initialized with two square brackets [][]. The first bracket specifies the size of an array, and the second bracket specifies the dimensions of the array which is going to be stored.

The following example declares jagged arrays.

Example: Jagged Arrays

int[][] jArray1 = new int[2][]; // can include two single-dimensional arrays

int[][,] jArray2 = new int[3][,]; // can include three two-dimensional arrays

II

n the above example, jArray1 can store up to two single-dimensional arrays. jArray2 can store up to three two-dimensional, arrays [,] specifies the two-dimensional array.

Example: Jagged Array

int[][] jArray = new int[2][];

jArray[0] = new int[3]{1, 2, 3};

jArray[1] = new int[4]{4, 5, 6, 7 };

You can also initialize a jagged array upon declaration like the below.

Example: Jagged Array

int[][] jArray = new int[2][]{

new int[3]{1, 2, 3},

new int[4]{4, 5, 6, 7}

};

jArray[0][0]; //returns 1

jArray[0][1]; //returns 2

jArray[0][2]; //returns 3

jArray[1][0]; //returns 4

jArray[1][1]; //returns 5

jArray[1][2]; //returns 6

jArray[1][3]; //returns 7

You can access a jagged array using two for loops, as shown below.

Example: Jagged Array

int[][] jArray = new int[2][]{

new int[3]{1, 2, 3},

new int[4]{4, 5, 6, 7}

};

for(int i=0; i<jArray.Length; i++)

{

for(int j=0; j < (jArray[i]).Length; j++)

Console.WriteLine(jArray[i][j]);

}

Console.WriteLine(jArray[i][j]);

}

The following jagged array stores two-dimensional arrays where the second bracket [,] indicates the two-dimensional array.

Example: Jagged Array

int[][,] jArray = new int[2][,];

jArray[0] = new int[3, 2] { { 1, 2 }, { 3, 4 }, { 5, 6 } };

jArray[1] = new int[2, 2] { { 7, 8 }, { 9, 10 } };

jArray[0][1, 1]; //returns 4

jArray[1][1, 0]; //returns 9

jArray[1][1, 1]; //returns 10

If you add one more bracket then it will be array of array of arry.

Example: Jagged Array

int[][][] intJaggedArray = new int[2][][]

{

new int[2][]

{

new int[3] { 1, 2, 3},

new int[2] { 4, 5}

},

new int[1][]

{

new int[3] { 7, 8, 9}

}

};

Console.WriteLine(intJaggedArray[0][0][0]); // 1

Console.WriteLine(intJaggedArray[0][1][1]); // 5

Console.WriteLine(intJaggedArray[1][0][2]); // 9

In the above example of a jagged array, three brackets [][][] means an array of array of array. So, intJaggedArray will contain two elements, which means two arrays. Now, each of these arrays also contains an array (single-dimension). intJaggedArray[0][0][0] points to the first element of first inner array. intJaggedArray[1][0][2] points to the third element of the second inner array. The following figure illustrates this.

[Diagram

Description automatically generated](https://www.tutorialsteacher.com/Content/images/csharp/jagged-array.png)

# C# - Collections

C# includes specialized classes that store series of values or objects are called collections.

There are two types of collections available in C#: non-generic collections and generic collections.

The System.Collections namespace contains the non-generic collection types and System.Collections.Generic namespace includes generic collection types.

In most cases, it is recommended to use the generic collections because they perform faster than non-generic collections and also minimize exceptions by giving compile-time errors.

## Generic Collections

C# includes the following generic collection classes in the System.Collections.Generic namespace.

| Generic Collections | Description |
| --- | --- |
| List<T> | Generic List<T> contains elements of specified type. It grows automatically as you add elements in it. |
| Dictionary<TKey,TValue> | Dictionary<TKey,TValue> contains key-value pairs. |
| SortedList<TKey,TValue> | SortedList stores key and value pairs. It automatically adds the elements in ascending order of key by default. |
| Queue<T> | Queue<T> stores the values in FIFO style (First In First Out). It keeps the order in which the values were added. It provides an Enqueue() method to add values and a Dequeue() method to retrieve values from the collection. |
| Stack<T> | Stack<T> stores the values as LIFO (Last In First Out). It provides a Push() method to add a value and Pop() & Peek() methods to retrieve values. |
| Hashset<T> | Hashset<T> contains non-duplicate elements. It eliminates duplicate elements. |

## Non-generic Collections

| Non-generic Collections | Usage |
| --- | --- |
| ArrayList | ArrayList stores objects of any type like an array. However, there is no need to specify the size of the ArrayList like with an array as it grows automatically. |
| SortedList | SortedList stores key and value pairs. It automatically arranges elements in ascending order of key by default. C# includes both, generic and non-generic SortedList collection. |
| Stack | Stack stores the values in LIFO style (Last In First Out). It provides a Push() method to add a value and Pop() & Peek() methods to retrieve values. C# includes both, generic and non-generic Stack. |
| Queue | Queue stores the values in FIFO style (First In First Out). It keeps the order in which the values were added. It provides an Enqueue() method to add values and a Dequeue() method to retrieve values from the collection. C# includes generic and non-generic Queue. |
| Hashtable | Hashtable stores key and value pairs. It retrieves the values by comparing the hash value of the keys. |
| BitArray | BitArray manages a compact array of bit values, which are represented as Booleans, where true indicates that the bit is on (1) and false indicates the bit is off (0). |

## C# - Array List

In C#, the ArrayList is a non-generic collection of objects whose size increases dynamically. It is the same as Array except that its size increases dynamically.

An ArrayList can be used to add unknown data where you don't know the types and the size of the data.

### Create an ArrayList

The ArrayList class included in the System.Collections namespace. Create an object of the ArrayList using the new keyword.

Example: Create an ArrayList

using System.Collections;

ArrayList arlist = new ArrayList();

// or

var arlist = new ArrayList(); // recommended

## Adding Elements in ArrayList

Use the Add() method or object initializer syntax to add elements in an ArrayList.

An ArrayList can contain multiple null and duplicate values.

Example: Adding Elements in ArrayList

// adding elements using ArrayList.Add() method

var arlist1 = new ArrayList();

arlist1.Add(1);

arlist1.Add("Bill");

arlist1.Add(" ");

arlist1.Add(true);

arlist1.Add(4.5);

arlist1.Add(null);

// adding elements using object initializer syntax

var arlist2 = new ArrayList()

{

2, "Steve", " ", true, 4.5, null

};

};

Use the AddRange(ICollection c) method to add an entire Array, HashTable, SortedList, rrayList, BitArray, Queue, and Stack in the ArrayList.

Example: Adding Entire Array/ArrayList into ArrayList

var arlist1 = new ArrayList();

var arlist2 = new ArrayList()

{

1, "Bill", " ", true, 4.5, null

};

int[] arr = { 100, 200, 300, 400 };

Queue myQ = new Queue();

myQ.Enqueue("Hello");

myQ.Enqueue("World!");

arlist1.AddRange(arlist2); //adding arraylist in arraylist

arlist1.AddRange(arr); //adding array in arraylist

arlist1.AddRange(myQ); //adding Queue in arraylist

### Accessing an ArrayList

The ArrayList class implements the IList interface. So, elements can be accessed using indexer, in the same way as an array. Index starts from zero and increases by one for each subsequent element.

An explicit casting to the appropriate types is required, or use the var variable.

Example: Accessing Elements of ArrayList

var arlist = new ArrayList()

{

1,

"Bill",

300,

4.5f

};

//Access individual item using indexer

int firstElement = (int) arlist[0]; //returns 1

string secondElement = (string) arlist[1]; //returns "Bill"

//int secondElement = (int) arlist[1]; //Error: cannot cover string to int

//using var keyword without explicit casting

var firstElement = arlist[0]; //returns 1

var secondElement = arlist[1]; //returns "Bill"

//var fifthElement = arlist[5]; //Error: Index out of range

//update elements

arlist[0] = "Steve";

arlist[1] = 100;

//arlist[5] = 500; //Error: Index out of range

### Iterate an ArrayList

The ArrayList implements the ICollection interface that supports iteration of the collection types. So, use the foreach and the for loop to iterate an ArrayList. The Count property of an ArrayList returns the total number of elements in an ArrayList.

Example: Iterate ArrayList

ArrayList arlist = new ArrayList()

{

1,

"Bill",

300,

4.5F

};

foreach (var item in arlist)

Console.Write(item + ", "); //output: 1, Bill, 300, 4.5,

for(int i = 0 ; i < arlist.Count; i++)

Console.Write(arlist[i] + ", "); //output: 1, Bill, 300, 4.5,

### Insert Elements in ArrayList

Use the Insert() method to insert an element at the specified index into an ArrayList.

Signature: *void Insert(int index, Object value)*

Example: Insert Element in ArrayList

ArrayList arlist = new ArrayList()

{

1,

"Bill",

300,

4.5f

};

arlist.Insert(1, "Second Item");

foreach (var val in arlist)

Console.WriteLine(val);

Use the InsertRange() method to insert a collection in an ArrayList at the specfied index.

Signature: *Void InsertRange(int index, ICollection c)*

Example: Insert Collection in ArrayList

ArrayList arlist1 = new ArrayList()

{

100, 200, 600

};

ArrayList arlist2 = new ArrayList()

{

300, 400, 500

};

arlist1.InsertRange(2, arlist2);

foreach(var item in arlist1)

Console.Write(item + ", "); //output: 100, 200, 300, 400, 500, 600,

### Remove Elements from ArrayList

Use the Remove(), RemoveAt(), or RemoveRange methods to remove elements from an ArrayList.

Example: Remove Elements from ArrayList

ArrayList arList = new ArrayList()

{

1,

null,

"Bill",

300,

" ",

4.5f,

300,

};

arList.Remove(null); //Removes first occurance of null

arList.RemoveAt(4); //Removes element at index 4

arList.RemoveRange(0, 2);//Removes two elements starting from 1st item (0 index)

### Check Element in ArrayList

Use the Contains() method to determine whether the specified element exists in the ArrayList or not. It returns true if exists otherwise returns false.

Example: Check for Elements

ArrayList arList = new ArrayList()

{

1,

"Bill",

300,

4.5f,

300

};

Console.WriteLine(arList.Contains(300)); // true

Console.WriteLine(arList.Contains("Bill")); // true

Console.WriteLine(arList.Contains(10)); // false

Console.WriteLine(arList.Contains("Steve")); // false

It is not recommended to use the ArrayList class due to performance issue. Instead, use List<object> to store heterogeneous objects. To store data of same data type, use Generic List<T>.

### ArrayList Class

The following diagram illustrates the ArrayList class.

[Diagram

Description automatically generated](https://www.tutorialsteacher.com/Content/images/csharp/arraylist.png)

### ArrayList Properties

| Properties | Description |
| --- | --- |
| Capacity | Gets or sets the number of elements that the ArrayList can contain. |
| Count | Gets the number of elements actually contained in the ArrayList. |
| IsFixedSize | Gets a value indicating whether the ArrayList has a fixed size. |
| IsReadOnly | Gets a value indicating whether the ArrayList is read-only. |
| Item | Gets or sets the element at the specified index. |

### ArrayList Methods

| Methods | Description |
| --- | --- |
| Add()/AddRange() | Add() method adds single elements at the end of ArrayList. AddRange() method adds all the elements from the specified collection into ArrayList. |
| Insert()/InsertRange() | Insert() method insert a single elements at the specified index in ArrayList. InsertRange() method insert all the elements of the specified collection starting from specified index in ArrayList. |
| Remove()/RemoveRange() | Remove() method removes the specified element from the ArrayList. RemoveRange() method removes a range of elements from the ArrayList. |
| RemoveAt() | Removes the element at the specified index from the ArrayList. |
| Sort() | Sorts entire elements of the ArrayList. |
| Reverse() | Reverses the order of the elements in the entire ArrayList. |
| Contains | Checks whether specified element exists in the ArrayList or not. Returns true if exists otherwise false. |
| Clear | Removes all the elements in ArrayList. |
| CopyTo | Copies all the elements or range of elements to compitible Array. |
| GetRange | Returns specified number of elements from specified index from ArrayList. |
| IndexOf | Search specified element and returns zero based index if found. Returns -1 if element not found. |
| ToArray | Returns compitible array from an ArrayList. |

## C# - List

The List<T> List<T> is a collection of strongly typed objects that can be accessed by index and having methods for sorting, searching, and modifying list. It is the generic version of the ArrayList that comes under System.Collection.GenericSystem.Collection.Generic namespace.

### List<T> Characteristics

* List<T> List<T> equivalent of the ArrayList, which implements IList<T>.
* It comes under System.Collections.Generic namespace.
* List<T> can contain elements of the specified type. It provides compile-time type checking and doesn't perform boxing-unboxing because it is generic.
* Elements can be added using the Add(), AddRange() methods or collection-initializer syntax.
* Elements can be accessed by passing an index e.g. myList[0]. Indexes start from zero.
* List<T> performs faster and less error-prone than the ArrayList.

### Creating a List

The List<T> List<T> is a generic collection, so you need to specify a type parameter for the type of data it can store. The following example shows how to create list and add elements.

Example: Adding elements in List

List<int> primeNumbers = new List<int>();

primeNumbers.Add(1); // adding elements using add() method

primeNumbers.Add(3);

primeNumbers.Add(5);

primeNumbers.Add(7);

var cities = new List<string>();

cities.Add("New York");

cities.Add("London");

cities.Add("Mumbai");

cities.Add("Chicago");

cities.Add(null);// nulls are allowed for reference type list

//adding elements using collection-initializer syntax

var bigCities = new List<string>()

{

"New York",

"London",

"Mumbai",

"Chicago"

};

};

In the above example, List<int> primeNumbers = new List<int>(); creates a list of int type. In the same way, cities and bigCities are string type list. You can then add elements in a list using the Add() method or the collection-initializer syntax.

You can also add elements of the custom classes using the collection-initializer syntax. The following adds objects of the Student class in the List<Student>.

Example: Add Custom Class Objects in List

var students = new List<Student>() {

new Student(){ Id = 1, Name="Bill"},

new Student(){ Id = 2, Name="Steve"},

new Student(){ Id = 3, Name="Ram"},

new Student(){ Id = 4, Name="Abdul"}

};

};

### Adding an Array in a List

Use the AddRange() method to add all the elements from an array or another collection to List.

AddRange() signature: void AddRange(IEnumerable<T> collection)

Example: Add Arrays in List

string[] cities = new string[3]{ "Mumbai", "London", "New York" };

var popularCities = new List<string>();

// adding an array in a List

popularCities.AddRange(cities);

var favouriteCities = new List<string>();

// adding a List

favouriteCities.AddRange(popularCities);

favouriteCities.AddRange(popularCities);

### Accessing a List

A list can be accessed by an index, a for/foreach loop, and using LINQ queries. Indexes of a list start from zero. Pass an index in the square brackets to access individual list items, same as array. Use a foreach or for loop to iterate a List<T> collection.

Example: Accessing List

List<int> numbers = new List<int>() { 1, 2, 5, 7, 8, 10 };

Console.WriteLine(numbers[0]); // prints 1

Console.WriteLine(numbers[1]); // prints 2

Console.WriteLine(numbers[2]); // prints 5

Console.WriteLine(numbers[3]); // prints 7

// using foreach LINQ method

numbers.ForEach(num => Console.WriteLine(num + ", "));//prints 1, 2, 5, 7, 8, 10,

// using for loop

for(int i = 0; i < numbers.Count; i++)

Console.WriteLine(numbers[i]);

### Accessing a List using LINQ

The List<T> implements the IEnumerable interface. So, we can query a list using LINQ query syntax or method syntax, as shown below.

Example: LINQ Query on List

var students = new List<Student>() {

new Student(){ Id = 1, Name="Bill"},

new Student(){ Id = 2, Name="Steve"},

new Student(){ Id = 3, Name="Ram"},

new Student(){ Id = 4, Name="Abdul"}

};

//get all students whose name is Bill

var result = from s in students

where s.Name == "Bill"

select s;

foreach(var student in result)

Console.WriteLine(student.Id + ", " + student.Name);

### Insert Elements in List

Use the Insert() method inserts an element into the List<T> collection at the specified index.

Insert() signature:void Insert(int index, T item);

Example: Insert elements into List

var numbers = new List<int>(){ 10, 20, 30, 40 };

numbers.Insert(1, 11);// inserts 11 at 1st index: after 10.

foreach (var num in numbers)

Console.Write(num);

### Remove Elements from List

Use the Remove() method to remove the first occurrence of the specified element in the List<T> collection. Use the RemoveAt() method to remove an element from the specified index. If no element at the specified index, then the ArgumentOutOfRangeException will be thrown.

Remove() signature: bool Remove(T item)

RemoveAt() signature: void RemoveAt(int index)

Example: Remove elements from List

var numbers = new List<int>(){ 10, 20, 30, 40, 10 };

numbers.Remove(10); // removes the first 10 from a list

numbers.RemoveAt(2); //removes the 3rd element (index starts from 0)

//numbers.RemoveAt(10); //throws ArgumentOutOfRangeException

foreach (var el in intList)

Console.Write(el); //prints 20 30

### Check Elements in List

Use the Contains() method to determine whether an element is in the List<T> or not.

Example: Contains()

var numbers = new List<int>(){ 10, 20, 30, 40 };

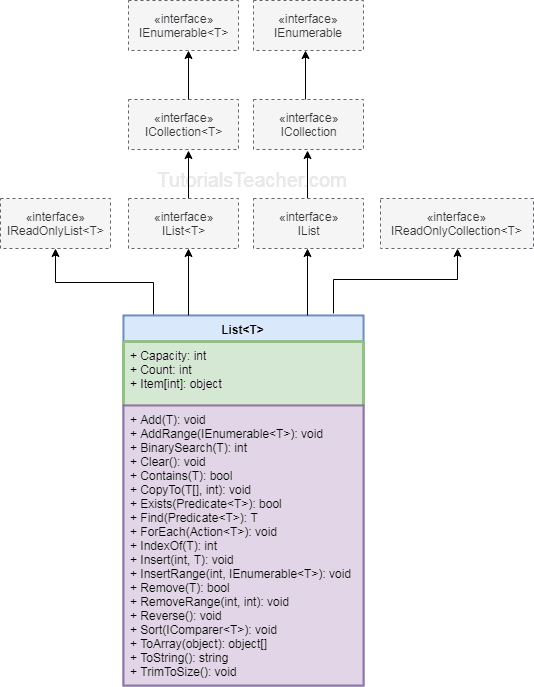
numbers.Contains(10); // returns true

numbers.Contains(11); // returns false

numbers.Contains(20); // returns true

### List<T> Class Hierarchy

The following diagram illustrates the List<T> hierarchy.

[](https://www.tutorialsteacher.com/Content/images/csharp/list.png)

### List<T> Class Properties and Methods

The following table lists the important properties and methods of List<T> class:

| Property | Usage |
| --- | --- |
| Items | Gets or sets the element at the specified index |
| Count | Returns the total number of elements exists in the List<T> |

| Method | Usage |
| --- | --- |
| Add | Adds an element at the end of a List<T>. |
| AddRange | Adds elements of the specified collection at the end of a List<T>. |
| BinarySearch | Search the element and returns an index of the element. |
| Clear | Removes all the elements from a List<T>. |
| Contains | Checks whether the specified element exists or not in a List<T>. |
| Find | Finds the first element based on the specified predicate function. |
| Foreach | Iterates through a List<T>. |
| Insert | Inserts an element at the specified index in a List<T>. |
| InsertRange | Inserts elements of another collection at the specified index. |
| Remove | Removes the first occurrence of the specified element. |
| RemoveAt | Removes the element at the specified index. |
| RemoveRange | Removes all the elements that match the supplied predicate function. |
| Sort | Sorts all the elements. |
| TrimExcess | Sets the capacity to the actual number of elements. |
| TrueForAll | Determines whether every element in the List<T> matches the conditions defined by the specified predicate. |

## C# - SortedList<TKey, TValue>

The SortedList<TKey, TValue>, and SortedList are collection classes that can store key-value pairs that are sorted by the keys based on the associated IComparer implementation. For example, if the keys are of primitive types, then sorted in ascending order of keys.

C# supports generic and non-generic SortedList. It is recommended to use generic SortedList<TKey, TValue> because it performs faster and less error-prone than the non-generic SortedList.

### SortedList Characteristics

* SortedList<TKey, TValue> SortedList<TKey, TValue> is an array of key-value pairs sorted by keys.
* Sorts elements as soon as they are added. Sorts primitive type keys in ascending order and object keys based on IComparer<T>.
* Comes under System.Collection.Generic namespace.
* A key must be unique and cannot be null.
* A value can be null or duplicate.
* A value can be accessed by passing associated key in the indexer mySortedList[key]
* Contains elements of type KeyValuePair<TKey, TValue>
* It uses less memory than SortedDictionary<TKey,TValue>.
* It is faster in the retrieval of data once sorted, whereas SortedDictionary<TKey, TValue> is faster in insertion and removing key-value pairs.

### Creating a SortedList

The following example demonstrates how to create a generic SortedList<TKey, TValue>, and add key-value pairs in it.

Example: Create a SortedList and Add Elements

//SortedList of int keys, string values

SortedList<int, string> numberNames = new SortedList<int, string>();

numberNames.Add(3, "Three");

numberNames.Add(1, "One");

numberNames.Add(2, "Two");

numberNames.Add(4, null);

numberNames.Add(10, "Ten");

numberNames.Add(5, "Five");

//The following will throw exceptions

//numberNames.Add("Three", 3); //Compile-time error: key must be int type

//numberNames.Add(1, "One"); //Run-time exception: duplicate key

//numberNames.Add(null, "Five");//Run-time exception: key cannot be null

In the above example, a generic SortedList<TKey, TValue> object is created by specifying the type of keys and values it is going to store. The SortedList<int, string> will store keys of int type and values of string type.

The Add() method is used to add a single key-value pair in a SortedList. Keys cannot be null or duplicate. If found, it will throw a run-time exception. Values can be duplicate and null if the type is nullable.

Use the collection-initializer syntax to initialize a SortedList with multiple key-value pairs at the time of instantiating, as shown below.

//Creating a SortedList of string keys, string values

//using collection-initializer syntax

SortedList<string,string> cities = new SortedList<string,string>()

{

{"London", "UK"},

{"New York", "USA"},

{ "Mumbai", "India"},

{"Johannesburg", "South Africa"}

};

};

The SortedList rearranges key-value pairs in the ascending order of keys as soon as a key-value pair added. The following example displays all the keys and values using foreach loop.

Example: SortedList Elements Default Sorting Order

SortedList<int,string> numberNames = new SortedList<int,string>()

{

{3, "Three"},

{5, "Five"},

{1, "One"}

};

Console.WriteLine("---Initial key-values--");

foreach(KeyValuePair<int, string> kvp in numberNames)

Console.WriteLine("key: {0}, value: {1}", kvp.Key , kvp.Value );

numberNames.Add(6, "Six");

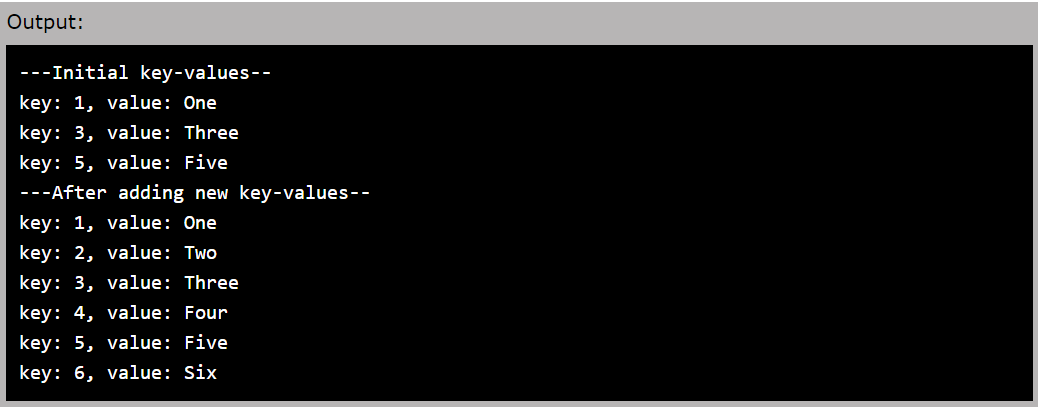
numberNames.Add(2, "Two");

numberNames.Add(4, "Four");

Console.WriteLine("---After adding new key-values--");

foreach(var kvp in numberNames)

Console.WriteLine("key: {0}, value: {1}", kvp.Key , kvp.Value );

Microsoft To Launch Cross-Platform Email Client Codenamed Project Monarch

### Accessing SortedList

Specify a key in the indexer sortedList[key], to get or set a value in the SortedList.

Example: Access SortedList Values

SortedList<int,string> numberNames = new SortedList<int,string>()

{

{3, "Three"},

{1, "One"},

{2, "Two"}

};

Console.WriteLine(numberNames[1]); //output: One

Console.WriteLine(numberNames[2]); //output: Two

Console.WriteLine(numberNames[3]); //output: Three

//Console.WriteLine(numberNames[10]); //run-time KeyNotFoundException

numberNames[2] = "TWO"; //updates value

numberNames[4] = "Four"; //adds a new key-value if a key does not exists

Above, numberNames[10] will throw a KeyNotFoundException because specified key 10 does not exist in a sortedlist. To prevent this exception, use ContainsKey() or TryGetValue() methods, as shown below.

Example: ContainsKey() and TryGetValue()

SortedList<int, string> numberNames = new SortedList<int,string>()

{

{3, "Three"},

{1, "One"},

{2, "Two"}

};

if(numberNames.ContainsKey(4)){

numberNames[4] = "four";

}

int result;

if(numberNames.TryGetValue(4, out result))

Console.WriteLine("Key: {0}, Value: {1}", 4, result);

Output:

Key:4, Value: Four

Use Keys and Values properties if you want to iterate a SortedList using a for loop.

Example: Iterate SortedList using For Loop

SortedList<int, string> numberNames = new SortedList<int,string>()

{

{3, "Three"},

{1, "One"},

{2, "Two"}

};

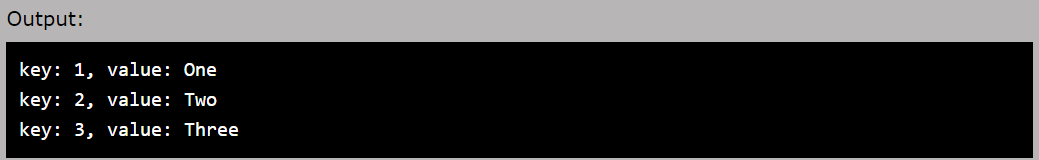
for (int i = 0; i < numberNames.Count; i++)

{

Console.WriteLine("key: {0}, value: {1}", numberNames.Keys[i], numberNames.Values[i]);

}

}



### Remove Elements from SortedList

Use the Remove(key) and RemoveAt(index) methods to remove key-value pairs from a SortedList.

Example: Remove Elements

SortedList<int,string> numberNames = new SortedList<int,string>()

{

{3, "Three"},

{1, "One"},

{2, "Two"},

{5, "Five"},

{4, "Four"}

};

numberNames.Remove(1);//removes key 1 pair

numberNames.Remove(10);//removes key 1 pair, no error if not exists

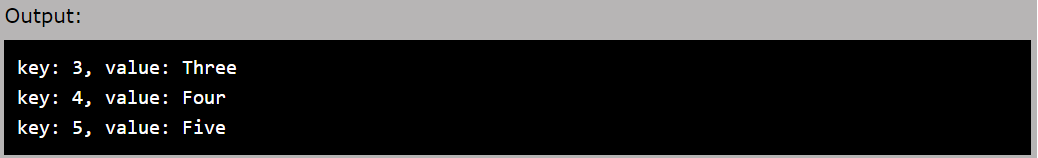
numberNames.RemoveAt(0);//removes key-value pair from index 0

//numberNames.RemoveAt(10);//run-time exception: ArgumentOutOfRangeException

foreach(var kvp in numberNames)

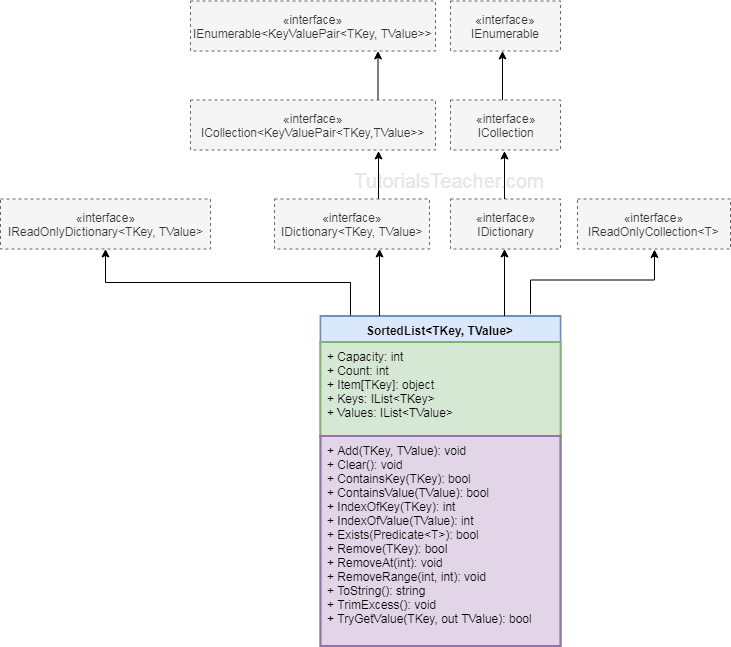
Console.WriteLine("key: {0}, value: {1}", kvp.Key , kvp.Value );

Console.WriteLine("key: {0}, value: {1}", kvp.Key , kvp.Value );



### SortedList Class Hierarchy

The following diagram illustrates the SortedList hierarchy.

[](https://www.tutorialsteacher.com/Content/images/csharp/sortedlist-generic.png)

## C# - Dictionary<TKey, TValue>

The Dictionary<TKey, TValue> is a generic collection that stores key-value pairs in no particular order.

### Dictionary Characteristics

* Dictionary<TKey, TValue>Dictionary<TKey, TValue> stores key-value pairs.
* Comes under System.Collections.Generic namespace.
* Implements IDictionary<TKey, TValue> interface.
* Keys must be unique and cannot be null.
* Values can be null or duplicate.
* Values can be accessed by passing associated key in the indexer e.g., myDictionary[key]
* Elements are stored as KeyValuePair<TKey, TValue> objects.

### Creating a Dictionary

You can create the Dictionary<TKey, TValue> object by passing the type of keys and values it can store. The following example shows how to create a dictionary and add key-value pairs.

Example: Create Dictionary and Add Elements

IDictionary<int, string> numberNames = new Dictionary<int, string>();

numberNames.Add(1,"One"); //adding a key/value using the Add() method

numberNames.Add(2,"Two");

numberNames.Add(3,"Three");

//The following throws run-time exception: key already added.

//numberNames.Add(3, "Three");

foreach(KeyValuePair<int, string> kvp in numberNames)

Console.WriteLine("Key: {0}, Value: {1}", kvp.Key, kvp.Value);

//creating a dictionary using collection-initializer syntax

var cities = new Dictionary<string, string>(){

{"UK", "London, Manchester, Birmingham"},

{"USA", "Chicago, New York, Washington"},

{"India", "Mumbai, New Delhi, Pune"}

};

foreach(var kvp in cities)

Console.WriteLine("Key: {0}, Value: {1}", kvp.Key, kvp.Value);

In the above example, numberNames is a Dictionary<int, string> type dictionary, so it can store int keys and string values. In the same way, cities is a Dictionary<string, string> type dictionary, so it can store string keys and string values. Dictionary cannot include duplicate or null keys, whereas values can be duplicated or null. Keys must be unique otherwise, it will throw a runtime exception.

### Access Dictionary Elements

The Dictionary can be accessed using indexer. Specify a key to get the associated value. You can also use the ElementAt() method to get a KeyValuePair from the specified index.

Example: Access Dictionary Elements

var cities = new Dictionary<string, string>(){

{"UK", "London, Manchester, Birmingham"},

{"USA", "Chicago, New York, Washington"},

{"India", "Mumbai, New Delhi, Pune"}

};

Console.WriteLine(cities["UK"]); //prints value of UK key

Console.WriteLine(cities["USA"]);//prints value of USA key

//Console.WriteLine(cities["France"]); // run-time exception: Key does not exist

//use ContainsKey() to check for an unknown key

if(cities.ContainsKey("France")){

Console.WriteLine(cities["France"]);

}

//use TryGetValue() to get a value of unknown key

string result;

if(cities.TryGetValue("France", out result))

{

Console.WriteLine(result);

}

//use ElementAt() to retrieve key-value pair using index

for (int i = 0; i < cities.Count; i++)

{

Console.WriteLine("Key: {0}, Value: {1}",

cities.ElementAt(i).Key,

cities.ElementAt(i).Value);

}

cities.ElementAt(i).Key,

cities.ElementAt(i).Value);

}

### Update Dictionary

Update the value of a key by specifying a key in the indexer. It will throw the KeyNotFoundException if a key does not exist in the dictionary, therefore use the ContainsKey() method before accessing unknown keys.

Example: Update Dictionary Elements

var cities = new Dictionary<string, string>(){

{"UK", "London, Manchester, Birmingham"},

{"USA", "Chicago, New York, Washington"},

{"India", "Mumbai, New Delhi, Pune"}

};

cities["UK"] = "Liverpool, Bristol"; // update value of UK key

cities["USA"] = "Los Angeles, Boston"; // update value of USA key

//cities["France"] = "Paris"; //throws run-time exception: KeyNotFoundException

if(cities.ContainsKey("France")){

cities["France"] = "Paris";

}

}

### Remove Elements in Dictionary

The Remove() method deletes an existing key-value pair from a dictionary. The Clear() method deletes all the elements of the dictionary.

Example: Remove Dictionary Elements

var cities = new Dictionary<string, string>(){

{"UK", "London, Manchester, Birmingham"},

{"USA", "Chicago, New York, Washington"},

{"India", "Mumbai, New Delhi, Pune"}

};

cities.Remove("UK"); // removes UK

//cities.Remove("France"); //throws run-time exception: KeyNotFoundException

if(cities.ContainsKey("France")){ // check key before removing it

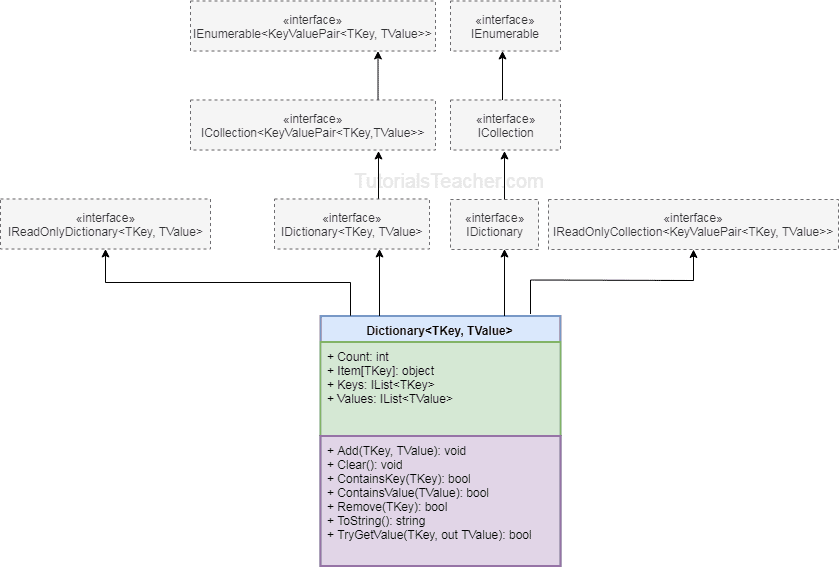
cities.Remove("France");

}

cities.Clear(); //removes all elements

### Dictionary Class Hierarchy

The following diagram illustrates the generic Dictionary class hierarchy.

[](https://www.tutorialsteacher.com/Content/images/csharp/generic-dictionary.png)

## C# - Hashtable

The Hashtable Hashtable is a non-generic collection that stores key-value pairs, similar to generic Dictionary<TKey, TValue> collection. It optimizes lookups by computing the hash code of each key and stores it in a different bucket internally and then matches the hash code of the specified key at the time of accessing values.

### Hashtable Characteristics

* HashtableHashtable stores key-value pairs.
* Comes under System.Collection namespace.
* Implements IDictionary interface.
* Keys must be unique and cannot be null.
* Values can be null or duplicate.
* Values can be accessed by passing associated key in the indexer e.g. myHashtable[key]myHashtable[key]
* Elements are stored as DictionaryEntry objects.

### Creating a Hashtable

The following example demonstrates creating a Hashtable and adding elements.

Example: Create and Add Elements

Hashtable numberNames = new Hashtable();

numberNames.Add(1,"One"); //adding a key/value using the Add() method

numberNames.Add(2,"Two");

numberNames.Add(3,"Three");

//The following throws run-time exception: key already added.

//numberNames.Add(3, "Three");

foreach(DictionaryEntry de in numberNames)

Console.WriteLine("Key: {0}, Value: {1}", de.Key, de.Value);

//creating a Hashtable using collection-initializer syntax

var cities = new Hashtable(){

{"UK", "London, Manchester, Birmingham"},

{"USA", "Chicago, New York, Washington"},

{"India", "Mumbai, New Delhi, Pune"}

};

foreach(DictionaryEntry de in cities)

Console.WriteLine("Key: {0}, Value: {1}", de.Key, de.Value);

The HashtableHashtable collection can include all the elements of Dictionary, as shown below.

Example: Add Dictionary in Hashtable

Dictionary<int, string> dict = new Dictionary<int, string>();

dict.Add(1, "one");

dict.Add(2, "two");

dict.Add(3, "three");

Hashtable ht = new Hashtable(dict);

### Update Hashtable

You can retrieve the value of an existing key from the Hashtable by passing a key in indexer. The Hashtable is a non-generic collection, so you must type cast values while retrieving it.

Example: Update Hashtable

//creating a Hashtable using collection-initializer syntax

var cities = new Hashtable(){

{"UK", "London, Manchester, Birmingham"},

{"USA", "Chicago, New York, Washington"},

{"India", "Mumbai, New Delhi, Pune"}

};

string citiesOfUK = (string) cities["UK"]; //cast to string

string citiesOfUSA = (string) cities["USA"]; //cast to string

Console.WriteLine(citiesOfUK);

Console.WriteLine(citiesOfUSA);

cities["UK"] = "Liverpool, Bristol"; // update value of UK key

cities["USA"] = "Los Angeles, Boston"; // update value of USA key

if(!cities.ContainsKey("France")){

cities["France"] = "Paris";

}

}

### Remove Elements in Hashtable

The Remove() method removes the key-value that match with the specified in the Hashtable. It throws the KeyNotfoundException if the specified key not found in the Hashtable, so check for an existing key using the ContainsKey() method before removing.

Use the Clear() method to remove all the elements in one shot.

Example: Remove Elements from Hashtable

var cities = new Hashtable(){

{"UK", "London, Manchester, Birmingham"},

{"USA", "Chicago, New York, Washington"},

{"India", "Mumbai, New Delhi, Pune"}

};

cities.Remove("UK"); // removes UK

//cities.Remove("France"); //throws run-time exception: KeyNotFoundException

if(cities.ContainsKey("France")){ // check key before removing it

cities.Remove("France");

}

cities.Clear(); //removes all elements

### Hashtable Class Hierarchy

The following diagram illustrates the Hashtable class hierarchy.

[Diagram

Description automatically generated](https://www.tutorialsteacher.com/Content/images/csharp/hashtable.png)

## C# - Stack<T>

StackStack is a special type of collection that stores elements in LIFO style (Last In First Out). C# includes the generic Stack<T> and non-generic Stack collection classes. It is recommended to use the generic Stack<T> collection.

Stack is useful to store temporary data in LIFO style, and you might want to delete an element after retrieving its value.

### Stack<T> Characteristics

* Stack<T>Stack<T> is Last In First Out collection.
* It comes under System.Collection.Generic namespace.
* Stack<T> can contain elements of the specified type. It provides compile-time type checking and doesn't perform boxing-unboxing because it is generic.
* Elements can be added using the Push() method. Cannot use collection-initializer syntax.
* Elements can be retrieved using the Pop() and the Peek() methods. It does not support an indexer.

### Creating a Stack

You can create an object of the Stack<T> by specifying a type parameter for the type of elements it can store. The following example creates and adds elements in the Stack<T> using the Push() method. Stack allows null (for reference types) and duplicate values.

Example: Create and Add Elements in Stack

Stack<int> myStack = new Stack<int>();

myStack.Push(1);

myStack.Push(2);

myStack.Push(3);

myStack.Push(4);

foreach (var item in myStack)

Console.Write(item + ","); //prints 4,3,2,1,

You can also create a Stack from an array, as shown below.

Example: Create and Add Elements in Stack

int[] arr = new int[]{ 1, 2, 3, 4};

Stack<int> myStack = new Stack<int>(arr);

foreach (var item in myStack)

Console.Write(item + ","); //prints 4,3,2,1,

### Stack<T> Properties and Methods:

| Property | Usage |
| --- | --- |
| Count | Returns the total count of elements in the Stack. |

| Method | Usage |
| --- | --- |
| Push(T) | Inserts an item at the top of the stack. |
| Peek() | Returns the top item from the stack. |
| Pop() | Removes and returns items from the top of the stack. |
| Contains(T) | Checks whether an item exists in the stack or not. |
| Clear() | Removes all items from the stack. |

### Pop()

The Pop() method returns the last element and removes it from a stack. If a stack is empty, then it will throw the InvalidOperationException. So, always check for the number of elements in a stack before calling the Pop() method.

Example: Access Stack using Pop()

Stack<int> myStack = new Stack<int>();

myStack.Push(1);

myStack.Push(2);

myStack.Push(3);

myStack.Push(4);

Console.Write("Number of elements in Stack: {0}", myStack.Count);

while (myStack.Count > 0)

Console.Write(myStack.Pop() + ",");

Console.Write("Number of elements in Stack: {0}", myStack.Count);



### Peek()

The Peek() method returns the lastly added value from the stack but does not remove it. Calling the Peek() method on an empty stack will throw the InvalidOperationException. So, always check for elements in the stack before retrieving elements using the Peek() method.

Example: Retrieve Elements usign Peek()

Stack<int> myStack = new Stack<int>();

myStack.Push(1);

myStack.Push(2);

myStack.Push(3);

myStack.Push(4);

Console.Write("Number of elements in Stack: {0}", myStack.Count);// prints 4

if(myStack.Count > 0){

Console.WriteLine(myStack.Peek()); // prints 4

Console.WriteLine(myStack.Peek()); // prints 4

}

Console.Write("Number of elements in Stack: {0}", myStack.Count);// prints 4

### Contains()

The Contains() Contains() method checks whether the specified element exists in a Stack collection or not. It returns true if it exists, otherwise false.

Example: Contains()

Stack<int> myStack = new Stack<int>();

myStack.Push(1);

myStack.Push(2);

myStack.Push(3);

myStack.Push(4);

myStack.Contains(2); // returns true

myStack.Contains(10); // returns false

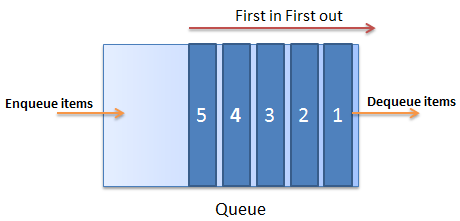
## C# - Queue<T>

QueueQueue is a special type of collection that stores the elements in FIFO style (First In First Out), exactly opposite of the Stack<T> collection. It contains the elements in the order they were added. C# includes generic Queue<T> and non-generic Queue collection. It is recommended to use the generic Queue<T> collection.

### Queue<T> Characteristics

* Queue<T>Queue<T> is FIFO (First In First Out) collection.
* It comes under System.Collection.Generic namespace.
* Queue<T> can contain elements of the specified type. It provides compile-time type checking and doesn't perform boxing-unboxing because it is generic.
* Elements can be added using the Enqueue() method. Cannot use collection-initializer syntax.
* Elements can be retrieved using the Dequeue() and the Peek() methods. It does not support an indexer.

The following figure illustrates the Queue collection:

[](https://www.tutorialsteacher.com/Content/images/csharp/csharp-queue.png)

### Creating a Queue

You can create an object of the Queue<T> by specifying a type parameter for the type of elements it can store. The following example creates and adds elements in the Queue<T> using the Enqueue() method. A Queue collection allows null (for reference types) and duplicate values.

Example: Create and Add Elements in the Queue

Queue<int> callerIds = new Queue<int>();

callerIds.Enqueue(1);

callerIds.Enqueue(2);

callerIds.Enqueue(3);

callerIds.Enqueue(4);

foreach(var id in callerIds)

Console.Write(id); //prints 1234

### Queue<T> Properties and Methods

| Property | Usage |
| --- | --- |
| Count | Returns the total count of elements in the Queue. |

| Method | Usage |
| --- | --- |
| Enqueue(T) | Adds an item into the queue. |
| Dequeue | Returns an item from the beginning of the queue and removes it from the queue. |
| Peek(T) | Returns an first item from the queue without removing it. |
| Contains(T) | Checks whether an item is in the queue or not |
| Clear() | Removes all the items from the queue. |

### Retrieve Elements from a Queue

The Dequeue() and the Peek() method is used to retrieve the first element in a queue collection. The Dequeue() removes and returns the first element from a queue because the queue stores elements in FIFO order. Calling the Dequeue() method on an empty queue will throw the InvalidOperation exception. So, always check that the total count of a queue is greater than zero before calling it.

Example: Reading Queue

Queue<string> strQ = new Queue<string>();

strQ.Enqueue("H");

strQ.Enqueue("e");

strQ.Enqueue("l");

strQ.Enqueue("l");

strQ.Enqueue("o");

Console.WriteLine("Total elements: {0}", strQ.Count); //prints 5

while (strQ.Count > 0)

Console.WriteLine(strQ.Dequeue()); //prints Hello

Console.WriteLine("Total elements: {0}", strQ.Count); //prints 0

The Peek() method always returns the first item from a queue collection without removing it from the queue. Calling the Peek() method on an empty queue will throw a run-time exception InvalidOperationException.

Example: Peek()

Queue<string> strQ = new Queue<string>();

strQ.Enqueue("H");

strQ.Enqueue("e");

strQ.Enqueue("l");

strQ.Enqueue("l");

strQ.Enqueue("o");

Console.WriteLine("Total elements: {0}", strQ.Count); //prints 5

if(strQ.Count > 0){

Console.WriteLine(strQ.Peek()); //prints H

Console.WriteLine(strQ.Peek()); //prints H

}

Console.WriteLine("Total elements: {0}", strQ.Count); //prints 5

### Contains()

The Contains() method checks whether an item exists in a queue or not. It returns true if the specified item exists, otherwise returns false.

Contains() Signature: bool Contains(object obj);

Example: Contains()

Queue<int> callerIds = new Queue<int>();

callerIds.Enqueue(1);

callerIds.Enqueue(2);

callerIds.Enqueue(3);

callerIds.Enqueue(4);

callerIds.Contains(2); //true

callerIds.Contains(10); //false

## C# - HashSet

In C#, HashSet is an unordered collection of unique elements. This collection is introduced in *.NET 3.5*. It supports the implementation of sets and uses the hash table for storage. This collection is of the generic type collection and it is defined under *System.Collections.Generic* namespace. It is generally used when we want to prevent duplicate elements from being placed in the collection. The performance of the HashSet is much better in comparison to the list.

Important Points:

* The HashSet class implements the ICollection, IEnumerable, IReadOnlyCollection, ISet, IEnumerable, IDeserializationCallback, and ISerializable interfaces.
* In HashSet, the order of the element is not defined. You cannot sort the elements of HashSet.
* In HashSet, the elements must be unique.
* In HashSet, duplicate elements are not allowed.
* Is provides many mathematical set operations, such as intersection, union, and difference.
* The capacity of a HashSet is the number of elements it can hold.
* A HashSet is a dynamic collection means the size of the HashSet is automatically increased when the new elements are added.
* In HashSet, you can only store the same type of elements.

### How to create a HashSet?

The HashSet class provides*7 different types of constructors* which are used to create a HashSet, here we only use *HashSet()*, constructor.

HashSet(): It is used to create an instance of the HashSet class that is empty and uses the default equality comparer for the set type.

Step 1: Include System.Collections.Generic namespace in your program with the help of using keyword:

using System.Collections.Generic;

Step 2: Create a HashSet using the HashSet class as shown below:

HashSet<Type\_of\_hashset> Hashset\_name = new HashSet<Type\_of\_hashset>();

Step 3: If you want to add elements in your HashSet, then use *Add()* method to add elements in your HashSet. And you can also store elements in your HashSet using collection initializer.

Step 4: The elements of HashSet is accessed by using a *foreach*loop. As shown in the below example.

**Example:**

// C# program to illustrate how to

// create hashset

using System;

using System.Collections.Generic;

class GFG {

// Main Method

static public void Main()

{

// Creating HashSet

// Using HashSet class

HashSet<string> myhash1 = new HashSet<string>();

// Add the elements in HashSet

// Using Add method

myhash1.Add("C");

myhash1.Add("C++");

myhash1.Add("C#");

myhash1.Add("Java");

myhash1.Add("Ruby");

Console.WriteLine("Elements of myhash1:");

// Accessing elements of HashSet

// Using foreach loop

foreach(var val in myhash1)

{

Console.WriteLine(val);

}

// Creating another HashSet

// using collection initializer

// to initialize HashSet

HashSet<int> myhash2 = new HashSet<int>() {10,

100,1000,10000,100000};

// Display elements of myhash2

Console.WriteLine("Elements of myhash2:");

foreach(var value in myhash2)

{

Console.WriteLine(value);

}

}

}

**Output**:

Elements of myhash1:

C

C++

C#

Java

Ruby

Elements of myhash2:

10

100

1000

10000

100000

### How to remove elements from the HashSet?

In HashSet, you are allowed to remove elements from the HashSet. HashSet<T> class provides three different methods to remove elements and the methods are:

* Remove(T): This method is used to remove the specified element from a HashSet object.
* RemoveWhere(Predicate): This method is used to remove all elements that match the conditions defined by the specified predicate from a HashSet collection.
* Clear: This method is used to remove all elements from a HashSet object.

**Example 1:**

// C# program to illustrate how to

// remove elements of HashSet

using System;

using System.Collections.Generic;

class GFG {

// Main Method

static public void Main()

{

// Creating HashSet

// Using HashSet class

HashSet<string> myhash = new HashSet<string>();

// Add the elements in HashSet

// Using Add method

myhash.Add("C");

myhash.Add("C++");

myhash.Add("C#");

myhash.Add("Java");

myhash.Add("Ruby");

// Before using Remove method

Console.WriteLine("Total number of elements present (Before Removal)"+

" in myhash: {0}", myhash.Count);

// Remove element from HashSet

// Using Remove method

myhash.Remove("Ruby");

// After using Remove method

Console.WriteLine("Total number of elements present (After Removal)"+

" in myhash: {0}", myhash.Count);

// Remove all elements from HashSet

// Using Clear method

myhash.Clear();

Console.WriteLine("Total number of elements present"+

" in myhash:{0}", myhash.Count);

}

}

**Output:**

Total number of elements present in myhash: 5

Total number of elements present in myhash: 4

Total number of elements present in myhash:0

### Set Operations

HashSet class also provides some methods that are used to perform different operations on sets and the methods are:

* [UnionWith(IEnumerable)](https://www.geeksforgeeks.org/c-union-of-two-hashset/): This method is used to modify the current HashSet object to contain all elements that are present in itself, the specified collection, or both.  
  Example:

// C# program to illustrate set operations

using System;

using System.Collections.Generic;

class GFG {

static public void Main()

{

// Creating HashSet

// Using HashSet class

HashSet<string> myhash1 = new HashSet<string>();

// Add the elements in HashSet

// Using Add method

myhash1.Add("C");

myhash1.Add("C++");

myhash1.Add("C#");

myhash1.Add("Java");

myhash1.Add("Ruby");

// Creating another HashSet

// Using HashSet class

HashSet<string> myhash2 = new HashSet<string>();

// Add the elements in HashSet

// Using Add method

myhash2.Add("PHP");

myhash2.Add("C++");

myhash2.Add("Perl");

myhash2.Add("Java");

// Using UnionWith method

myhash1.UnionWith(myhash2);

foreach(var ele in myhash1)

{

Console.WriteLine(ele);

}

}

}

**Output**:

C

C++

C#

Java

Ruby

PHP

Perl

* [IntersectWith(IEnumerable)](https://www.geeksforgeeks.org/c-intersection-of-two-hashsets/): This method is used to modify the current HashSet object to contain only elements that are present in that object and in the specified collection.  
  Example:

// C# program to illustrate set operations

using System;

using System.Collections.Generic;

class GFG {

// Main Method

static public void Main()

{

// Creating HashSet

// Using HashSet class

HashSet<string> myhash1 = new HashSet<string>();

// Add the elements in HashSet

// Using Add method

myhash1.Add("C");

myhash1.Add("C++");

myhash1.Add("C#");

myhash1.Add("Java");

myhash1.Add("Ruby");

// Creating another HashSet

// Using HashSet class

HashSet<string> myhash2 = new HashSet<string>();

// Add the elements in HashSet

// Using Add method

myhash2.Add("PHP");

myhash2.Add("C++");

myhash2.Add("Perl");

myhash2.Add("Java");

// Using IntersectWith method

myhash1.IntersectWith(myhash2);

foreach(var ele in myhash1)

{

Console.WriteLine(ele);

}

}

}

**Output**:

C++

Java

* [ExceptWith(IEnumerable)](https://www.geeksforgeeks.org/c-remove-all-elements-in-a-collection-from-a-hashset/): This method is used to remove all elements in the specified collection from the current HashSet object.

Example:

// C# program to illustrate set operations

using System;

using System.Collections.Generic;

class GFG {

// Main Method

static public void Main()

{

// Creating HashSet

// Using HashSet class

HashSet<string> myhash1 = new HashSet<string>();

// Add the elements in HashSet

// Using Add method

myhash1.Add("C");

myhash1.Add("C++");

myhash1.Add("C#");

myhash1.Add("Java");

myhash1.Add("Ruby");

// Creating another HashSet

// Using HashSet class

HashSet<string> myhash2 = new HashSet<string>();

// Add the elements in HashSet

// Using Add method

myhash2.Add("PHP");

myhash2.Add("C++");

myhash2.Add("Perl");

myhash2.Add("Java");

// Using ExceptWith method

myhash1.ExceptWith(myhash2);

foreach(var ele in myhash1)

{

Console.WriteLine(ele);

}

}

}

**Output**:

C

C#

Ruby

Let us see an example to remove duplicate strings using C# HashSet.

**Example:**

using System;

using System.Collections.Generic;

using System.Linq;

class Program {

   static void Main() {

      string[] arr1 = {"Table","Chair","Pen","Clip","Table"};

      Console.WriteLine(string.Join(",", arr1));

      // HashSet

      var h = new HashSet<string>(arr1);

      // eliminates duplicate words

      string[] arr2 = h.ToArray();

      Console.WriteLine(string.Join(",", arr2));

   }

}

# C# - Type Conversions (Type Casting)

Type conversion happens when we assign the value of one data type to another. If the data types are compatible, then C# does Automatic Type Conversion. If not comparable, then they need to be converted explicitly which is known as Explicit Type conversion. For example, assigning an int value to a long variable.

## Implicit Type Casting / Automatic Type Conversion

It happens when:

* The two data types are compatible.
* When we assign value of a smaller data type to a bigger data type.

For Example, in C#, the numeric data types are compatible with each other but no automatic conversion is supported from numeric type to char or boolean. Also, char and boolean are not compatible with each other. Before converting, the compiler first checks the compatibility according to the following figure and then it decides whether it is alright or there some error.

**Following table shows the implicit types of conversion that is supported by C# :**

|  |  |
| --- | --- |
| Convert from Data Type | **Convert to Data Type** |
| byte | short, int, long, float, double |
| short | int, long, float, double |
| int | long, float, double |
| long | float, double |
| float | double |

**Example**:

// C# program to demonstrate the

// Implicit Type Conversion

using System;

namespace Casting{

class GFG {

// Main Method

public static void Main(String []args)

{

int i = 57;

// automatic type conversion

long l = i;

// automatic type conversion

float f = l;

// Display Result

Console.WriteLine("Int value " +i);

Console.WriteLine("Long value " +l);

Console.WriteLine("Float value " +f);

}

}

}

**Output**:

Int value 57

Long value 57

Float value 57

## Explicit Type Casting

There may be compilation error when types not compatible with each other. For example, assigning double value to int data type:

// C# program to illustrate incompatible data

// type for explicit type conversion

using System;

namespace Casting{

class GFG {

// Main Method

public static void Main(String []args)

{

double d = 765.12;

// Incompatible Data Type

int i = d;

// Display Result

Console.WriteLine("Value of i is ", +i);

}

}

}

**Error**:

prog.cs(14,21): error CS0266: Cannot implicitly convert type `double' to `int'.

An explicit conversion exists (are you missing a cast?)

So, if we want to assign a value of larger data type to a smaller data type we perform explicit type casting.

* This is useful for incompatible data types where automatic conversion cannot be done.
* Here, target-type specifies the desired type to convert the specified value to.
* Sometimes, it may result into the lossy conversion.

**Example**:

// C# program to demonstrate the

// Explicit Type Conversion

using System;

namespace Casting{

class GFG {

// Main Method

public static void Main(String []args)

{

double d = 765.12;

// Explicit Type Casting

int i = (int)d;

// Display Result

Console.WriteLine("Value of i is " +i);

}

}

}

**Output**:

Value of i is 765

**Explanation**:   
Here due to lossy conversion, the value of i becomes 765 and there is a loss of 0.12 value.

C# provides built-in methods for Type-Conversions as follows:

|  |  |  |
| --- | --- | --- |
| **Method** |  | **Description** |
| ToBoolean |  | It will converts a type to Boolean value |
| ToChar |  | It will converts a type to a character value |
| ToByte |  | It will converts a value to Byte Value |
| ToDecimal |  | It will converts a value to Decimal point value |
| ToDouble |  | It will converts a type to double data type |
| ToInt16 |  | It will converts a type to 16-bit integer |
| ToInt32 |  | It will converts a type to 32 bit integer |
| ToInt64 |  | It will converts a type to 64 bit integer |
| ToString |  | It will converts a given type to string |
| ToUInt16 |  | It will converts a type to unsigned 16 bit integer |
| ToUInt32 |  | It will converts a type to unsigned 32 bit integer |
| ToUInt64 |  | It will converts a type to unsigned 64 bit integer |

**Example**:

// C# program to demonstrate the

// Built- In Type Conversion Methods

using System;

namespace Casting{

class GFG {

// Main Method

public static void Main(String []args)

{

int i = 12;

double d = 765.12;

float f = 56.123F;

// Using Built- In Type Conversion

// Methods & Displaying Result

Console.WriteLine(Convert.ToString(f));

Console.WriteLine(Convert.ToInt32(d));

Console.WriteLine(Convert.ToUInt32(f));

Console.WriteLine(Convert.ToDouble(i));

Console.WriteLine("AjaySingala");

}

}

}

**Output**:

56.123

765

56

12

AjaySingala

# C# - Serialization

Serialization is the process of converting an object into a stream of bytes to store the object or transmit it to memory, a database, or a file. Its main purpose is to save the state of an object to be able to recreate it when needed. The reverse process is called deserialization.

Often, we need to store objects to a physical storage so it can be read back and converted back to an object. The process of storing an object to a physical storage is called serialization. The process of reading a serialized object back into memory is deserialization.

**Note**: Serialization and Deserialization are also known as Marshal and Unmarshal respectively.

In simple words serialization in C# is a process of storing the object instance to a persistent storage. Serialization stores state of objects i.e., member variable values to persistent storage such as a disk. Deserialization is reverse of serialization. It is a process of reading objects from a file where they have been stored. In this code sample we will see how to serialize and deserialize objects using C#.

## How serialization works

This illustration shows the overall process of serialization:



The object is serialized to a stream that carries the data. The stream may also have information about the object's type, such as its version, culture, and assembly name. From that stream, the object can be stored in a database, a file, or memory.

## Namespaces involved

Following namespaces are involved in serialization process,

* System.Runtime.Serialization
* System.Runtime.Serialization.Formatter
* System.Runtime.Serialization.Formatters.Binary
* System.Text.Json
* System.Text.Json.Serialization
* System.Xml.Serialization

## Uses for serialization

Serialization allows the developer to save the state of an object and re-create it as needed, providing storage of objects as well as data exchange. Through serialization, a developer can perform actions such as:

* Sending the object to a remote application by using a web service
* Passing an object from one domain to another
* Passing an object through a firewall as a JSON or XML string
* Maintaining security or user-specific information across applications

## JSON serialization

The System.Text.Json namespace contains classes for JavaScript Object Notation (JSON) serialization and deserialization. JSON is an open standard that is commonly used for sharing data across the web.

JSON serialization serializes the public properties of an object into a string, byte array, or stream that conforms to the RFC 8259 JSON specification. To control the way JsonSerializer serializes or deserializes an instance of the class:

* Use a JsonSerializerOptions object
* Apply attributes from the System.Text.Json.Serialization namespace to classes or properties
* Implement custom converters

## Binary and XML serialization

The System.Runtime.Serialization namespace contains classes for binary and XML serialization and deserialization.

Binary serialization uses binary encoding to produce compact serialization for uses such as storage or socket-based network streams. In binary serialization, all members, even members that are read-only, are serialized, and performance is enhanced.

**Warning:** Binary serialization can be dangerous.

XML serialization serializes the public fields and properties of an object, or 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 XML. System.Xml.Serialization contains classes for serializing and deserializing XML. You apply attributes to classes and class members to control the way the XmlSerializer serializes or deserializes an instance of the class.

## Making an object serializable

For binary or XML serialization, you need:

* The object to be serialized
* A stream to contain the serialized object
* A System.Runtime.Serialization.Formatter instance

Apply the SerializableAttribute attribute to a type to indicate that instances of the type can be serialized. An exception is thrown if you attempt to serialize but the type doesn't have the SerializableAttribute attribute.

To prevent a field from being serialized, apply the NonSerializedAttribute attribute. If a field of a serializable type contains a pointer, a handle, or some other data structure that is specific to a particular environment, and the field cannot be meaningfully reconstituted in a different environment, then you may want to make it nonserializable.

If a serialized class contains references to objects of other classes that are marked SerializableAttribute, those objects will also be serialized.

## Basic and custom serialization

Binary and XML serialization can be performed in two ways, basic and custom.

Basic serialization uses .NET to automatically serialize the object. The only requirement is that the class has the SerializableAttribute attribute applied. The NonSerializedAttribute can be used to keep specific fields from being serialized.

When you use basic serialization, the versioning of objects may create problems. You would use custom serialization when versioning issues are important. Basic serialization is the easiest way to perform serialization, but it does not provide much control over the process.

In custom serialization, you can specify exactly which objects will be serialized and how it will be done. The class must be marked SerializableAttribute and implement the ISerializable interface. If you want your object to be deserialized in a custom manner as well, use a custom constructor.

## Example 1: Binary Serialization

using System;

using System.IO;

using System.Runtime.Serialization;

using System.Runtime.Serialization.Formatters.Binary;

public class SerialTest {

public void SerializeNow() {

ClassToSerialize c = new ClassToSerialize();

File f = new File("temp.dat");

Stream s = f.Open(FileMode.Create);

BinaryFormatter b = new BinaryFormatter();

b.Serialize(s, c);

s.Close();

}

public void DeSerializeNow() {

ClassToSerialize c = new ClassToSerialize();

File f = new File("temp.dat");

Stream s = f.Open(FileMode.Open);

BinaryFormatter b = new BinaryFormatter();

c = (ClassToSerialize) b.Deserialize(s);

Console.WriteLine(c.name);

s.Close();

}

public static void Main(string[] s) {

SerialTest st = new SerialTest();

st.SerializeNow();

st.DeSerializeNow();

}

}

public class ClassToSerialize {

public int age = 100;

public string name = "bipin";

}

### Explanation

Here we have our own class named ClassToSerialize. This class has two public valiables name and age with some default values. We will write this class to a disk file (temp.dat) using SerializeTest class.

SerializeTest class has two methods SerializeNow() and DeSerializeNow() which perform the task of serialization and deserialization respectively.

The general steps for serializing are,

* Create an instance of File that will store serialized object.
* Create a stream from the file object.
* Create an instance of BinaryFormatter.
* Call serialize method of the instance passing it stream and object to serialize.

The steps for de-serializing the object are similar. The only change is that you need to call deserialize method of BinaryFormatter object.

## Example 2: Class / Object Serialization

Now, let us see an example where we have used 'real' class with public and shared members and properties to encapsulate them. The class also uses another supporting class. This is just to make clear that if your class contains further classes, all the classes in the chain will be serialized.

using System;

using System.IO;

using System.Runtime.Serialization;

using System.Runtime.Serialization.Formatters.Binary;

public class SerialTest {

public void SerializeNow() {

ClassToSerialize c = new ClassToSerialize();

c.Name = "bipin";

c.Age = 26;

ClassToSerialize.CompanyName = "xyz";

File f = new File("temp.dat");

Stream s = f.Open(FileMode.Create);

BinaryFormatter b = new BinaryFormatter();

b.Serialize(s, c);

s.Close();

}

public void DeSerializeNow() {

ClassToSerialize c = new ClassToSerialize();

File f = new File("temp.dat");

Stream s = f.Open(FileMode.Open);

BinaryFormatter b = new BinaryFormatter();

c = (ClassToSerialize) b.Deserialize(s);

Console.WriteLine("Name :" + c.Name);

Console.WriteLine("Age :" + c.Age);

Console.WriteLine("Company Name :" + ClassToSerialize.CompanyName);

Console.WriteLine("Company Name :" + c.GetSupportClassString());

s.Close();

}

public static void Main(string[] s) {

SerialTest st = new SerialTest();

st.SerializeNow();

st.DeSerializeNow();

}

}

public class ClassToSerialize {

private int age;

private string name;

static string companyname;

SupportClass supp = new SupportClass();

public ClassToSerialize() {

supp.SupportClassString = "In support class";

}

public int Age {

get {

return age;

}

set {

age = value;

}

}

public string Name {

get {

return name;

}

set {

name = value;

}

}

public static string CompanyName {

get {

return companyname;

}

set {

companyname = value;

}

}

public string GetSupportClassString() {

return supp.SupportClassString;

}

}

public class SupportClass {

public string SupportClassString;

}

## Example 3: Array Serialization

This example shows how to serialize array of objects.

using System;

using System.IO;

using System.Runtime.Serialization;

using System.Runtime.Serialization.Formatters.Binary;

public class SerialTest {

public void SerializeNow() {

ClassToSerialize[] c = new ClassToSerialize[3];

c[0] = new ClassToSerialize();

c[0].Name = "bipin";

c[0].Age = 26;

c[1] = new ClassToSerialize();

c[1].Name = "abc";

c[1].Age = 75;

c[2] = new ClassToSerialize();

c[2].Name = "pqr";

c[2].Age = 50;

ClassToSerialize.CompanyName = "xyz";

File f = new File("temp.dat");

Stream s = f.Open(FileMode.Create);

BinaryFormatter b = new BinaryFormatter();

b.Serialize(s, c);

s.Close();

}

public void DeSerializeNow() {

ClassToSerialize[] c;

File f = new File("temp.dat");

Stream s = f.Open(FileMode.Open);

BinaryFormatter b = new BinaryFormatter();

c = (ClassToSerialize[]) b.Deserialize(s);

Console.WriteLine("Name :" + c[2].Name);

Console.WriteLine("Age :" + c[2].Age);

Console.WriteLine("Company Name :" + ClassToSerialize.CompanyName);

s.Close();

}

public static void Main(string[] s) {

SerialTest st = new SerialTest();

st.SerializeNow();

st.DeSerializeNow();

}

}

public class ClassToSerialize {

private int age;

private string name;

static string companyname;

public int Age {

get {

return age;

}

set {

age = value;

}

}

public string Name {

get {

return name;

}

set {

name = value;

}

}

public static string CompanyName {

get {

return companyname;

}

set {

companyname = value;

}

}

}

## Example 4: Serialize Class / Object

using System;

using System.IO;

using System.Linq;

using System.Runtime.Serialization;

using System.Runtime.Serialization.Formatters.Binary;

using System.Text;

using System.Threading.Tasks;

namespace DemoApplication

{

[Serializable]

class Tutorial

{

public int ID;

public String Name;

static void Main(string[] args)

{

Tutorial obj = new Tutorial();

obj.ID = 1;

obj.Name = ".Net";

IFormatter formatter = new BinaryFormatter();

Stream stream = new FileStream(@"E:\ExampleNew.txt",FileMode.Create,FileAccess.Write);

formatter.Serialize(stream, obj);

stream.Close();

stream = new FileStream(@"E:\ExampleNew.txt",FileMode.Open,FileAccess.Read);

Tutorial objnew = (Tutorial)formatter.Deserialize(stream);

Console.WriteLine(objnew.ID);

Console.WriteLine(objnew.Name);

Console.ReadKey();

}

}

}

### Code Explanation:

1. The class which needs to be serialized needs to have the [Serializable] attribute. This is a keyword in C#. This keyword is then attached to the Tutorial class. If you don’t mention this attribute, you will get an error when you try to serialize the class.
2. Next is the definition of the class which will be serialized. Here we are defining a class called “Tutorial” and providing 2 properties, one is “ID” and the other is “Name.”
3. First, we create an object of the Tutorial class. We then assign the value of “1” to ID and “.net” to the name property.
4. We then use the formatter class which is used to serialize or convert the object to a binary format. The data in the file in serialization is done in binary format. Next, we create a file stream object. The file stream object is used to open the file Example.txt for writing purposes. The keywords FileMode.Create and FileMode.Write is used to specifically mention that the file should be opened for writing purposes.
5. Finally, we use the Serialize method to transfer the binary data to the file. We then close the stream since the write operation is complete.
6. Ensure that the data is present in the file, we use deserialization to deserialize the object from the file.
7. We create the object “stream” to open the file Example.txt in reading only mode.
8. We then use the formatter class which is used to deserialize the object, which is stored in the Example.txt file. The object returned is set to the object objnew.
9. Finally, we display the properties of the object “objnew” to the console using the “ID” and “name” properties.

## Serialize and Deserialize JSON

Let’s see how to use the System.Text.Json namespace to serialize to and deserialize from JavaScript Object Notation (JSON).

The code samples here:

* Use the library directly, not through a framework such as ASP.NET Core.
* Use the JsonSerializer class with custom types to serialize from and deserialize into.
* Use the WriteIndented option to format the JSON for human readability when that is helpful.

For production use, you would typically accept the default value of false for this setting, since adding unnecessary whitespace may incur a negative impact on performance and bandwidth usage.

## Namespaces

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 namespaces:

using System.Text.Json;

using System.Text.Json.Serialization;

**Important:** Attributes from the **System.Runtime.Serialization** namespace aren't supported in System.Text.Json.

## Serialize (Write) .NET objects as JSON

To write JSON to a string or to a file, call the JsonSerializer.Serialize method.

### JSON Serialization

The following example creates JSON as a string:

using System;

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.

### Serialize JSON to a File

The following example uses synchronous code to create a JSON file:

using System;

using System.IO;

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"}

### Serialize JSON to a File Asynchronously

The following example uses asynchronous code to create a JSON file:

using System;

using System.IO;

using System.Text.Json;

using System.Threading.Tasks;

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"}

### Serialize JSON using Generics

The preceding examples use type inference for the type being serialized. An overload of Serialize() takes a generic type parameter:

using System;

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 of Class with Collections and User Defined Types

Here's an example showing how a class that contains collection properties and a user-defined type is serialized:

using System;

using System.Collections.Generic;

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"

// ]

//}

## Deserialize (Read) JSON as .NET Objects

A common way to deserialize JSON is to first create a 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, you pass the type of the class you created as the generic type parameter. For the non-generic overloads, you pass the type of the class you created as a method parameter. You can deserialize either synchronously or asynchronously. Any JSON properties that aren't represented in your class are ignored.

### Deserialize a JSON String

The following example shows how to deserialize a JSON string:

using System;

using System.Collections.Generic;

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

### Deserialize JSON from a File

To deserialize from a file by using synchronous code, read the file into a string, as shown in the following example:

using System;

using System.IO;

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 =

await 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

### Deserialize JSON from a File Asynchronously

To deserialize from a file by using asynchronous code, call the DeserializeAsync method:

using System;

using System.IO;

using System.Text.Json;

using System.Threading.Tasks;

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

**Tip**

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:

* 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 in .NET 6, 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 the **Utf8JsonReader** directly.
* Use Visual Studio 2019 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.

## Serialize to Formatted JSON

To pretty-print the JSON output, set JsonSerializerOptions.WriteIndented to true:

using System;

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"

//}

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

## Include fields

Use the JsonSerializerOptions.IncludeFields global setting or the [JsonInclude] attribute to include fields when serializing or deserializing, as shown in the following example:

using System;

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.

## 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:

using System;

using System.Net.Http;

using System.Net.Http.Json;

using System.Threading.Tasks;

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.

## Serialize and Deserialize XML

### Serialize (Read) Data from an XML File

This example reads object data that was previously written to an XML file using the XmlSerializer class.

**Example:**

public class Book

{

public String title;

}

public void ReadXML()

{

// First write something so that there is something to read ...

var b = new Book { title = "Serialization Overview" };

var writer = new System.Xml.Serialization.XmlSerializer(typeof(Book));

var wfile =

new System.IO.StreamWriter(@"c:\temp\SerializationOverview.xml");

writer.Serialize(wfile, b);

wfile.Close();

// Now we can read the serialized book ...

System.Xml.Serialization.XmlSerializer reader =

new System.Xml.Serialization.XmlSerializer(typeof(Book));

System.IO.StreamReader file = new System.IO.StreamReader(

@"c:\temp\SerializationOverview.xml");

Book overview = (Book)reader.Deserialize(file);

file.Close();

Console.WriteLine(overview.title);

}

#### Compiling the Code

Replace the file name "c:\temp\SerializationOverview.xml" with the name of the file containing the serialized data. For more information about serializing data, see [How to write object data to an XML file (C#)](#_Deserializing_a_C#) below.

The class must have a public constructor without parameters.

Only public properties and fields are deserialized.

#### Robust Programming

The following conditions may cause an exception:

* The class being serialized does not have a public, parameterless constructor.
* The data in the file does not represent data from the class to be deserialized.
* The file does not exist (IOException).

### Deserialize (Write) Data to an XML File

This example writes the object from a class to an XML file using the XmlSerializer class.

**Example:**

public class XMLWrite

{

static void Main(string[] args)

{

WriteXML();

}

public class Book

{

public String title;

}

public static void WriteXML()

{

Book overview = new Book();

overview.title = "Serialization Overview";

System.Xml.Serialization.XmlSerializer writer =

new System.Xml.Serialization.XmlSerializer(typeof(Book));

var path = Environment.GetFolderPath(Environment.SpecialFolder.MyDocuments) + "//SerializationOverview.xml";

System.IO.FileStream file = System.IO.File.Create(path);

writer.Serialize(file, overview);

file.Close();

}

}

#### Compiling the Code

The class being serialized must have a public constructor without parameters.

#### Robust Programming

The following conditions may cause an exception:

* The class being serialized does not have a public, parameterless constructor.
* The file exists and is read-only (IOException).
* The path is too long (PathTooLongException).
* The disk is full (IOException).

#### A Better Example

#### Serializing XML to C# Object

Let's understand how to convert an XML file into a C# object. Take note of the below small XML file to demonstrate.

<?xml version="1.0" encoding="utf-8"?>

<Company xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.w3.org/2001/XMLSchema">

  <Employee name="x" age="30" />

  <Employee name="y" age="32" />

</Company>

To convert this XML into an object, first you need to create a similar class structure in C#.

[XmlRoot(ElementName = "Company")]

**public** **class** Company

{

**public** Company()

    {

        Employees = **new** List<Employee>();

    }

    [XmlElement(ElementName = "Employee")]

**public** List<Employee> Employees { **get**; **set**; }

**public** Employee **this**[**string** name]

    {

**get** { **return** Employees.FirstOrDefault(s => **string**.Equals(s.Name, name, StringComparison.OrdinalIgnoreCase)); }

    }

}

**public** **class** Employee

{

    [XmlAttribute("name")]

**public** **string** Name { **get**; **set**; }

    [XmlAttribute("age")]

**public** **string** Age { **get**; **set**; }

}

Your XML and C# objects are ready. Let's see the final step of converting XML into a C# object. To do that, you need to use System.Xml.Serialization.XmlSerializer to serialize it.

**public** T DeserializeToObject<T>(**string** filepath) where T : **class**

{

    System.Xml.Serialization.XmlSerializer ser = **new** System.Xml.Serialization.XmlSerializer(**typeof**(T));

**using** (StreamReader sr = **new** StreamReader(filepath))

    {

**return** (T)ser.Deserialize(sr);

    }

}

Use the XML file path and use this function. You should see that the XML is converted into a company object with two employee objects.

#### Deserializing a C# Object into XML

Create a C# object, such as a company with a few employees, and then convert it into an XML file.

var company = **new** Company();

company.Employees = **new** List<Employee>() { **new** Employee() { Name = "o", Age = "10"}};

SerializeToXml(company, xmlFilePath);

**public** **static** **void** SerializeToXml<T>(T anyobject, **string** xmlFilePath)

{

    XmlSerializer xmlSerializer = **new** XmlSerializer(anyobject.GetType());

**using** (StreamWriter writer = **new** StreamWriter(xmlFilePath))

    {

        xmlSerializer.Serialize(writer, anyobject);

    }

}

The output should look like the below text after converting it into XML.

<?xml version="1.0" encoding="UTF-8"?>

<Company xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

  <Employee age="10" name="o"/>

</Company>

#### Another XML Example

using System;

public class clsPerson

{

public string FirstName;

public string MI;

public string LastName;

}

class class1

{

static void Main(string[] args)

{

clsPerson p=new clsPerson();

p.FirstName = "Jeff";

p.MI = "A";

p.LastName = "Price";

System.Xml.Serialization.XmlSerializer x = new System.Xml.Serialization.XmlSerializer(p.GetType());

x.Serialize(Console.Out, p);

Console.WriteLine();

Console.ReadLine();

}

}

##### Verification

To verify that your project works, press CTRL+F5 to run the project. A clsPerson object is created and populated with the values that you entered. This state is serialized to XML. The console window shows the following code:

<?xml version="1.0" encoding="IBM437"?>

<clsPerson xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.w3.org/2001/XMLSchema">

<FirstName>Jeff</FirstName>

<MI>A</MI>

<LastName>Price</LastName>

</clsPerson>

#### 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.

public class PurchaseOrder

{

public Address MyAddress;

}

public class Address

{

public string FirstName;

}

The serialized XML output might resemble the following.

<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.

public class PurchaseOrder

{

public Item [] ItemsOrders;

}

public class Item

{

public string ItemID;

public decimal ItemPrice;

}

The serialized class instance might resemble the following, if two items are ordered.

<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 use the XmlSerializer to serialize instances of these classes. Note that 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 will not be serialized. The class must include an Add method and an Item property (C# indexer) to be serialized.

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]{

get{return (Employee) empArray[index];}

}

public void CopyTo(Array a, int index){

empArray.CopyTo(a, index);

}

public int Count{

get{return empArray.Count;}

}

public object SyncRoot{

get{return this;}

}

public bool IsSynchronized{

get{return false;}

}

public IEnumerator GetEnumerator(){

return 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 – Exercise

You can cut and paste the following example code into a text file renamed with a .cs file name extension. Use the C# 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, and the same file is then read back 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 is used to serialize and deserialize the PurchaseOrder. Note that the code passes the type of the class that will be serialized to the constructor. The code also creates a FileStream that is 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 the 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 that you can also read the PO.xml file that is created to see the actual XML output.

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.ToLongDateString();

// 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\t Subtotal\t" + po.SubTotal +

"\n\t\t\t\t\t Shipping\t" + po.ShipCost +

"\n\t\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 +"\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 resemble the following:

<?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>

# C# - OOP

## OOP Principles

* Abstraction
* Encapsulation
* Polymorphism
* Inheritance
* Overriding
* Overloading
* Abstract
* Interface

## C# Class Members

* Fields
* Properties
* Methods
* Indexers

### Indexers

C# indexers are usually known as smart arrays. A C# indexer is a class property that allows you to access a member variable of a class or struct using the features of an array. In C#, indexers are created using this keyword. Indexers in C# are applicable on both classes and structs.

Defining an indexer allows you to create a class like that can allows its items to be accessed an array.  Instances of that class can be accessed using the [] array access operator.

#### Creating an Indexer

<modifier> <**return** type> **this** [argument list]

{

**get**

{

// your get block code

}

**set**

{

// your set block code

}

}

In the above code:

**<modifier>** can be private, public, protected or internal.

**<return type>** can be any valid C# types.

**this** his is a special keyword in C# to indicate the object of the current class.

**[argument list]** The formal-argument-list specifies the parameters of the indexer.

**Important points to remember on indexers:**

* Indexers are always created with **this** keyword.
* Parameterized property are called indexer.
* Indexers are implemented through get and set accessors for the [ ] operator.
* ref and out parameter modifiers are not permitted in indexer.
* The formal parameter list of an indexer corresponds to that of a method and at least one parameter should be specified.
* Indexer is an instance member so can't be static but property can be static.
* Indexers are used on group of elements.
* Indexer is identified by its signature where as a property is identified it's name.
* Indexers are accessed using indexes where as properties are accessed by names.
* Indexer can be overloaded.

Indexer are defined in pretty much same way as properties, with get and set functions. The main difference is that the name of the indexer is the keyword this.

Indexers are commonly used for classes, which represents some data structure, an array, list, map and so on.

Following program demonstrates how to use an indexer.

**using** System;

**namespace** Indexer\_example1

{

**class** Program

    {

**class** IndexerClass

        {

**private** **string**[] names = **new** **string**[10];

**public** **string** **this**[**int** i]

            {

**get**

                {

**return** names[i];

                }

**set**

                {

                    names[i] = value;

                }

            }

        }

**static** **void** Main(**string**[] args)

        {

            IndexerClass Team = **new** IndexerClass();

            Team[0] = "Rocky";

            Team[1] = "Teena";

            Team[2] = "Ana";

            Team[3] = "Victoria";

            Team[4] = "Yani";

            Team[5] = "Mary";

            Team[6] = "Gomes";

            Team[7] = "Arnold";

            Team[8] = "Mike";

            Team[9] = "Peter";

**for** (**int** i = 0; i < 10; i++)

            {

                Console.WriteLine(Team[i]);

            }

            Console.ReadKey();

        }

    }

}

### Difference between Indexers and Properties

|  |  |
| --- | --- |
| Indexers | Properties |
| Indexers are created with this keyword. | Properties don't require this keyword. |
| Indexers are identified by signature. | Properties are identified by their names. |
| Indexers are accessed using indexes. | Properties are accessed by their names. |
| Indexer are instance member, so can't be static. | Properties can be static as well as instance members. |
| A get accessor of an indexer has the same formal parameter list as the indexer. | A get accessor of a property has no parameters. |
| A set accessor of an indexer has the same formal parameter list as the indexer, in addition to the value parameter. | A set accessor of a property contains the implicit value parameter. |

# C# - Access Modifiers

**Keywords**:

* private
* public
* protected
* internal

## protected

A protected member is accessible within its class and by derived class instances.

A protected member of a base class is accessible in a derived class only if the access occurs through the derived class type. For example, consider the following code segment.

**Example 1**:

class A

{

protected int x = 123;

}

class B : A

{

static void Main()

{

var a = new A();

var b = new B();

// Error CS1540, because x can only be accessed by

// classes derived from A.

// a.x = 10;

// OK, because this class derives from A.

b.x = 10;

}

}

The statement a.x = 10 generates an error because it is made within the static method Main, and not an instance of class B.

Struct members cannot be protected because the struct cannot be inherited.

**Example 2**:

In this example, the class DerivedPoint is derived from Point. Therefore, you can access the protected members of the base class directly from the derived class.

class Point

{

protected int x;

protected int y;

}

class DerivedPoint: Point

{

static void Main()

{

var dpoint = new DerivedPoint();

// Direct access to protected members.

dpoint.x = 10;

dpoint.y = 15;

Console.WriteLine($"x = {dpoint.x}, y = {dpoint.y}");

}

}

// Output: x = 10, y = 15

If you change the access levels of x and y to private, the compiler will issue the error messages:

'Point.y' is inaccessible due to its protection level.

'Point.x' is inaccessible due to its protection level.

## internal

The internal keyword is an access modifier for types and type members.

This section covers internal access. The internal keyword is also part of the protected internal access modifier.

Internal types or members are accessible only within files in the same assembly, as in this example:

public class BaseClass

{

// Only accessible within the same assembly.

internal static int x = 0;

}

A common use of internal access is in component-based development because it enables a group of components to cooperate in a private manner without being exposed to the rest of the application code. For example, a framework for building graphical user interfaces could provide Control and Form classes that cooperate by using members with internal access. Since these members are internal, they are not exposed to code that is using the framework.

It is an error to reference a type or a member with internal access outside the assembly within which it was defined.

**Example 1**:

This example contains two files, Assembly1.cs and Assembly1\_a.cs. The first file contains an internal base class, BaseClass. In the second file, an attempt to instantiate BaseClass will produce an error.

// Assembly1.cs

// Compile with: /target:library

internal class BaseClass

{

public static int intM = 0;

}

C#Copy

// Assembly1\_a.cs

// Compile with: /reference:Assembly1.dll

class TestAccess

{

static void Main()

{

var myBase = new BaseClass(); // CS0122

}

}

**Example 2**:

In this example, use the same files you used in example 1, and change the accessibility level of BaseClass to public. Also change the accessibility level of the member intM to internal. In this case, you can instantiate the class, but you cannot access the internal member.

// Assembly2.cs

// Compile with: /target:library

public class BaseClass

{

internal static int intM = 0;

}

C#Copy

// Assembly2\_a.cs

// Compile with: /reference:Assembly2.dll

public class TestAccess

{

static void Main()

{

var myBase = new BaseClass(); // Ok.

BaseClass.intM = 444; // CS0117

}

}

# C# - Modifiers

## Override

The override modifier is required to extend or modify the abstract or virtual implementation of an inherited method, property, indexer, or event.

In the following example, the Square class must provide an overridden implementation of GetArea because GetArea is inherited from the abstract Shape class:

abstract class Shape

{

public abstract int GetArea();

}

class Square : Shape

{

private int \_side;

public Square(int n) => \_side = n;

// GetArea method is required to avoid a compile-time error.

public override int GetArea() => \_side \* \_side;

static void Main()

{

var sq = new Square(12);

Console.WriteLine($"Area of the square = {sq.GetArea()}");

}

}

// Output: Area of the square = 144

An override method provides a new implementation of the method inherited from a base class. The method that is overridden by an override declaration is known as the overridden base method. An override method must have the same signature as the overridden base method. Beginning with C# 9.0, override methods support covariant return types. In particular, the return type of an override method can derive from the return type of the corresponding base method. In C# 8.0 and earlier, the return types of an override method and the overridden base method must be the same.

You cannot override a non-virtual or static method. The overridden base method must be virtual, abstract, or override.

An override declaration cannot change the accessibility of the virtual method. Both the override method and the virtual method must have the same access level modifier.

You cannot use the new, static, or virtual modifiers to modify an override method.

An overriding property declaration must specify exactly the same access modifier, type, and name as the inherited property. Beginning with C# 9.0, read-only overriding properties support covariant return types. The overridden property must be virtual, abstract, or override.

**Example**:

This example defines a base class named Employee, and a derived class named SalesEmployee. The SalesEmployee class includes an extra field, salesbonus, and overrides the method CalculatePay in order to take it into account.

class TestOverride

{

public class Employee

{

public string Name { get; }

// Basepay is defined as protected, so that it may be

// accessed only by this class and derived classes.

protected decimal \_basepay;

// Constructor to set the name and basepay values.

public Employee(string name, decimal basepay)

{

Name = name;

\_basepay = basepay;

}

// Declared virtual so it can be overridden.

public virtual decimal CalculatePay()

{

return \_basepay;

}

}

// Derive a new class from Employee.

public class SalesEmployee : Employee

{

// New field that will affect the base pay.

private decimal \_salesbonus;

// The constructor calls the base-class version, and

// initializes the salesbonus field.

public SalesEmployee(string name, decimal basepay, decimal salesbonus)

: base(name, basepay)

{

\_salesbonus = salesbonus;

}

// Override the CalculatePay method

// to take bonus into account.

public override decimal CalculatePay()

{

return \_basepay + \_salesbonus;

}

}

static void Main()

{

// Create some new employees.

var employee1 = new SalesEmployee("Alice", 1000, 500);

var employee2 = new Employee("Bob", 1200);

Console.WriteLine($"Employee1 {employee1.Name} earned: {employee1.CalculatePay()}");

Console.WriteLine($"Employee2 {employee2.Name} earned: {employee2.CalculatePay()}");

}

}

/\*

Output:

Employee1 Alice earned: 1500

Employee2 Bob earned: 1200

\*/

## Readonly

The readonly keyword is a modifier that can be used in four contexts:

* In a field declaration, readonly indicates that assignment to the field can only occur as part of the declaration or in a constructor in the same class. A readonly field can be assigned and reassigned multiple times within the field declaration and constructor.

A readonly field can't be assigned after the constructor exits. This rule has different implications for value types and reference types:

* Because value types directly contain their data, a field that is a readonly value type is immutable.
* Because reference types contain a reference to their data, a field that is a readonly reference type must always refer to the same object. That object isn't immutable. The readonly modifier prevents the field from being replaced by a different instance of the reference type. However, the modifier doesn't prevent the instance data of the field from being modified through the read-only field.
* In a readonly struct type definition, readonly indicates that the structure type is immutable. For more information, see the readonly struct section of the Structure types article.
* In an instance member declaration within a structure type, readonly indicates that an instance member doesn't modify the state of the structure.
* In a ref readonly method return, the readonly modifier indicates that method returns a reference and writes aren't allowed to that reference.

The readonly struct and ref readonly contexts were added in C# 7.2. readonly struct members were added in C# 8.0

**Readonly field example**:

In this example, the value of the field year can't be changed in the method ChangeYear, even though it's assigned a value in the class constructor:

class Age

{

private readonly int \_year;

Age(int year)

{

\_year = year;

}

void ChangeYear()

{

//\_year = 1967; // Compile error if uncommented.

}

}

You can assign a value to a readonly field only in the following contexts:

* When the variable is initialized in the declaration, for example:

public readonly int y = 5;

* In an instance constructor of the class that contains the instance field declaration.
* In the static constructor of the class that contains the static field declaration.

These constructor contexts are also the only contexts in which it's valid to pass a readonly field as an out or ref parameter.

**Note**:

The readonly keyword is different from the **const** keyword. A const field can only be initialized at the declaration of the field. A readonly field can be assigned multiple times in the field declaration and in any constructor. Therefore, readonly fields can have different values depending on the constructor used. Also, while a const field is a compile-time constant, the readonly field can be used for run-time constants as in the following example:

public static readonly uint timeStamp = (uint)DateTime.Now.Ticks;

public class SamplePoint

{

public int x;

// Initialize a readonly field

public readonly int y = 25;

public readonly int z;

public SamplePoint()

{

// Initialize a readonly instance field

z = 24;

}

public SamplePoint(int p1, int p2, int p3)

{

x = p1;

y = p2;

z = p3;

}

public static void Main()

{

SamplePoint p1 = new SamplePoint(11, 21, 32); // OK

Console.WriteLine($"p1: x={p1.x}, y={p1.y}, z={p1.z}");

SamplePoint p2 = new SamplePoint();

p2.x = 55; // OK

Console.WriteLine($"p2: x={p2.x}, y={p2.y}, z={p2.z}");

}

/\*

Output:

p1: x=11, y=21, z=32

p2: x=55, y=25, z=24

\*/

}

In the preceding example, if you use a statement like the following example:

p2.y = 66; // Error

you'll get the compiler error message:

**A readonly field cannot be assigned to (except in a constructor or a variable initializer)**

## Static

Use the static modifier to declare a static member, which belongs to the type itself rather than to a specific object. The static modifier can be used to declare static classes. In classes, interfaces, and structs, you may add the static modifier to fields, methods, properties, operators, events, and constructors. The static modifier can't be used with indexers or finalizers.

Beginning with C# 8.0, you can add the static modifier to a local function. A static local function can't capture local variables or instance state.

Beginning with C# 9.0, you can add the static modifier to a lambda expression or anonymous method. A static lambda or anonymous method can't capture local variables or instance state.

**Example - static class**:

The following class is declared as static and contains only static methods:

static class CompanyEmployee

{

public static void DoSomething() { /\*...\*/ }

public static void DoSomethingElse() { /\*...\*/ }

}

A constant or type declaration is implicitly a static member. A static member can't be referenced through an instance. Instead, it's referenced through the type name. For example, consider the following class:

public class MyBaseC

{

public struct MyStruct

{

public static int x = 100;

}

}

To refer to the static member x, use the fully qualified name, MyBaseC.MyStruct.x, unless the member is accessible from the same scope:

Console.WriteLine(MyBaseC.MyStruct.x);

While an instance of a class contains a separate copy of all instance fields of the class, there's only one copy of each static field.

It isn't possible to use this to reference static methods or property accessors.

If the static keyword is applied to a class, all the members of the class must be static.

Classes, interfaces, and static classes may have static constructors. A static constructor is called at some point between when the program starts, and the class is instantiated.

**Example - static field and method**:

This example reads the name and ID of a new employee, increments the employee counter by one, and displays the information for the new employee and the new number of employees. This program reads the current number of employees from the keyboard.

public class Employee4

{

public string id;

public string name;

public Employee4()

{

}

public Employee4(string name, string id)

{

this.name = name;

this.id = id;

}

public static int employeeCounter;

public static int AddEmployee()

{

return ++employeeCounter;

}

}

class MainClass : Employee4

{

static void Main()

{

Console.Write("Enter the employee's name: ");

string name = Console.ReadLine();

Console.Write("Enter the employee's ID: ");

string id = Console.ReadLine();

// Create and configure the employee object.

Employee4 e = new Employee4(name, id);

Console.Write("Enter the current number of employees: ");

string n = Console.ReadLine();

Employee4.employeeCounter = Int32.Parse(n);

Employee4.AddEmployee();

// Display the new information.

Console.WriteLine($"Name: {e.name}");

Console.WriteLine($"ID: {e.id}");

Console.WriteLine($"New Number of Employees: {Employee4.employeeCounter}");

}

}

/\*

Input:

Matthias Berndt

AF643G

15

\*

Sample Output:

Enter the employee's name: Matthias Berndt

Enter the employee's ID: AF643G

Enter the current number of employees: 15

Name: Matthias Berndt

ID: AF643G

New Number of Employees: 16

\*/

**Example - static initialization**:

This example shows that you can initialize a static field by using another static field that is not yet declared. The results will be undefined until you explicitly assign a value to the static field.

class Test

{

static int x = y;

static int y = 5;

static void Main()

{

Console.WriteLine(Test.x);

Console.WriteLine(Test.y);

Test.x = 99;

Console.WriteLine(Test.x);

}

}

/\*

Output:

0

5

99

\*/

## Virtual

The virtual keyword is used to modify a method, property, indexer, or event declaration and allow for it to be overridden in a derived class. For example, this method can be overridden by any class that inherits it:

public virtual double Area()

{

return x \* y;

}

The implementation of a virtual member can be changed by an overriding member in a derived class.

When a virtual method is invoked, the run-time type of the object is checked for an overriding member. The overriding member in the most derived class is called, which might be the original member, if no derived class has overridden the member.

By default, methods are non-virtual. You cannot override a non-virtual method.

You cannot use the virtual modifier with the static, abstract, private, or override modifiers. The following example shows a virtual property:

class MyBaseClass

{

// virtual auto-implemented property. Overrides can only

// provide specialized behavior if they implement get and set accessors.

public virtual string Name { get; set; }

// ordinary virtual property with backing field

private int \_num;

public virtual int Number

{

get { return \_num; }

set { \_num = value; }

}

}

class MyDerivedClass : MyBaseClass

{

private string \_name;

// Override auto-implemented property with ordinary property

// to provide specialized accessor behavior.

public override string Name

{

get

{

return \_name;

}

set

{

if (!string.IsNullOrEmpty(value))

{

\_name = value;

}

else

{

\_name = "Unknown";

}

}

}

}

Virtual properties behave like virtual methods, except for the differences in declaration and invocation syntax.

* It is an error to use the virtual modifier on a static property.
* A virtual inherited property can be overridden in a derived class by including a property declaration that uses the override modifier.

**Example**:

In this example, the Shape class contains the two coordinates x, y, and the Area() virtual method. Different shape classes such as Circle, Cylinder, and Sphere inherit the Shape class, and the surface area is calculated for each figure. Each derived class has its own override implementation of Area().

Notice that the inherited classes Circle, Sphere, and Cylinder all use constructors that initialize the base class, as shown in the following declaration.

public Cylinder(double r, double h): base(r, h) {}

The following program calculates and displays the appropriate area for each figure by invoking the appropriate implementation of the Area() method, according to the object that is associated with the method.

class TestClass

{

public class Shape

{

public const double PI = Math.PI;

protected double \_x, \_y;

public Shape()

{

}

public Shape(double x, double y)

{

\_x = x;

\_y = y;

}

public virtual double Area()

{

return \_x \* \_y;

}

}

public class Circle : Shape

{

public Circle(double r) : base(r, 0)

{

}

public override double Area()

{

return PI \* \_x \* \_x;

}

}

public class Sphere : Shape

{

public Sphere(double r) : base(r, 0)

{

}

public override double Area()

{

return 4 \* PI \* \_x \* \_x;

}

}

public class Cylinder : Shape

{

public Cylinder(double r, double h) : base(r, h)

{

}

public override double Area()

{

return 2 \* PI \* \_x \* \_x + 2 \* PI \* \_x \* \_y;

}

}

static void Main()

{

double r = 3.0, h = 5.0;

Shape c = new Circle(r);

Shape s = new Sphere(r);

Shape l = new Cylinder(r, h);

// Display results.

Console.WriteLine("Area of Circle = {0:F2}", c.Area());

Console.WriteLine("Area of Sphere = {0:F2}", s.Area());

Console.WriteLine("Area of Cylinder = {0:F2}", l.Area());

}

}

/\*

Output:

Area of Circle = 28.27

Area of Sphere = 113.10

Area of Cylinder = 150.80

\*/

# C# - Data Types

## Anonymous Types

In C#, an anonymous type is a type (class) without any name that can contain public read-only properties only. It cannot contain other members, such as fields, methods, events, etc.

You create an anonymous type using the *new* operator with an object initializer syntax. The implicitly typed variable- var is used to hold the reference of anonymous types.

The following example demonstrates creating an anonymous type variable student that contains three properties named Id, FirstName, and LastName.

**Example**: Anonymous Type

var student = new { Id = 1, FirstName = "James", LastName = "Bond" };

The properties of anonymous types are read-only and cannot be initialized with a null, anonymous function, or a pointer type. The properties can be accessed using dot (.) notation, same as object properties. However, you cannot change the values of properties as they are read-only.

**Example**: Access Anonymous Type

var student = new { Id = 1, FirstName = "James", LastName = "Bond" };

Console.WriteLine(student.Id); //output: 1

Console.WriteLine(student.FirstName); //output: James

Console.WriteLine(student.LastName); //output: Bond

student.Id = 2;//Error: cannot chage value

student.FirstName = "Steve";//Error: cannot chage value

An anonymous type's property can include another anonymous type.

**Example**: Nested Anonymous Type

var student = new {

Id = 1,

FirstName = "James",

LastName = "Bond",

Address = new { Id = 1, City = "London", Country = "UK" }

};

You can create an array of anonymous types also.

**Example**: Array of Anonymous Types

var students = new[] {

new { Id = 1, FirstName = "James", LastName = "Bond" },

new { Id = 2, FirstName = "Steve", LastName = "Jobs" },

new { Id = 3, FirstName = "Bill", LastName = "Gates" }

};

An anonymous type will always be local to the method where it is defined. It cannot be returned from the method. However, an anonymous type can be passed to the method as object type parameter, but it is not recommended. If you need to pass it to another method, then use struct or class instead of an anonymous type.

Mostly, anonymous types are created using the Select clause of a LINQ queries to return a subset of the properties from each object in the collection.

**Example**: LINQ Query returns an Anonymous Type

class Program

{

static void Main(string[] args)

{

IList<Student> studentList = new List<Student>() {

new Student() { StudentID = 1, StudentName = "John", age = 18 },

new Student() { StudentID = 2, StudentName = "Steve", age = 21 },

new Student() { StudentID = 3, StudentName = "Bill", age = 18 },

new Student() { StudentID = 4, StudentName = "Ram" , age = 20 },

new Student() { StudentID = 5, StudentName = "Ron" , age = 21 }

};

var students = from s in studentList

select new { Id = s.StudentID, Name = s.StudentName };

foreach(var stud in students)

Console.WriteLine(stud.Id + "-" + stud.Name);

}

}

public class Student

{

public int StudentID { get; set; }

public string StudentName { get; set; }

public int age { get; set; }

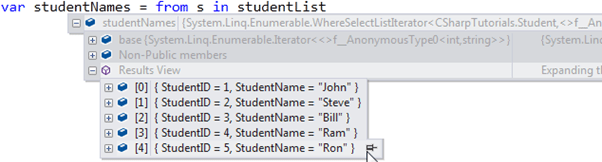
}

Shape, rectangle

Description automatically generated

In the above example, a select clause in the LINQ query selects only StudentID and StudentName properties and renames it to Id and Name, respectively. Thus, it is useful in saving memory and unnecessary code. The query result collection includes only StudentID and StudentName properties, as shown in the following debug view.

Visual Studio supports IntelliSense for anonymous types, as shown below.

[](https://www.tutorialsteacher.com/Content/images/csharp/anonymoustype-debugview.png)Anonymous Type Intellisense Support in Visual Studio

Internally, all the anonymous types are directly derived from the System.Object class. The compiler generates a class with some auto-generated name and applies the appropriate type to each property based on the value expression. Although your code cannot access it. Use GetType() method to see the name.

**Example**: Internal Name of an Anonymous Type

static void Main(string[] args)

{

var student = new { Id = 1, FirstName = "James", LastName = "Bond" };

Console.WriteLine(student.GetType().ToString());

}

## Dynamic Types

C# 4.0 (.NET 4.5) introduced a new type called dynamic that avoids compile-time type checking. A dynamic type escapes type checking at compile-time; instead, it resolves type at run time.

A dynamic type variables are defined using the dynamic keyword.

**Example**: dynamic Variable

dynamic MyDynamicVar = 1;

The compiler compiles dynamic types into object types in most cases. However, the actual type of a dynamic type variable would be resolved at run-time.

**Example**: dynamic Type at run-time

dynamic MyDynamicVar = 1;

Console.WriteLine(MyDynamicVar.GetType());



Dynamic types change types at run-time based on the assigned value. The following example shows how a dynamic variable changes type based on assigned value.

**Example**: dynamic

static void Main(string[] args)

{

dynamic MyDynamicVar = 100;

Console.WriteLine("Value: {0}, Type: {1}", MyDynamicVar, MyDynamicVar.GetType());

MyDynamicVar = "Hello World!!";

Console.WriteLine("Value: {0}, Type: {1}", MyDynamicVar, MyDynamicVar.GetType());

MyDynamicVar = true;

Console.WriteLine("Value: {0}, Type: {1}", MyDynamicVar, MyDynamicVar.GetType());

MyDynamicVar = DateTime.Now;

Console.WriteLine("Value: {0}, Type: {1}", MyDynamicVar, MyDynamicVar.GetType());

}

Text

Description automatically generated

The dynamic type variables is converted to other types implicitly.

**Example**: dynamic Type Conversion

dynamic d1 = 100;

int i = d1;

d1 = "Hello";

string greet = d1;

d1 = DateTime.Now;

DateTime dt = d1;

### Methods and Parameters

If you assign a class object to the dynamic type, then the compiler would not check for correct methods and properties name of a dynamic type that holds the custom class object. Consider the following example.

**Example**: Calling Methods

class Program

{

static void Main(string[] args)

{

dynamic stud = new Student();

stud.DisplayStudentInfo(1, "Bill");// run-time error, no compile-time error

stud.DisplayStudentInfo("1");// run-time error, no compile-time error

stud.FakeMethod();// run-time error, no compile-time error

}

}

public class Student

{

public void DisplayStudentInfo(int id)

{

}

}

In the above example, the C# compiler does not check for the number of parameters, parameters type, or non-existent. It validates these things at run-time, and if it is not valid, then throws a run-time exception. Note that Visual Studio IntelliSense is not supported for the dynamic types. Note that Visual Studio IntelliSense is not supported for the dynamic types.

## Nullable Types

As you know, a value type cannot be assigned a null value. For example, *int i = null* will give you a compile time error.

C# 2.0 introduced nullable types that allow you to assign null to value type variables. You can declare nullable types using Nullable<t> where T is a type.

**Example**: Nullable type

Nullable<int> i = null;

A nullable type can represent the correct range of values for its underlying value type, plus an additional *null* value. For example, Nullable<int> can be assigned any value from -2147483648 to 2147483647, or a null value.

The Nullable types are instances of System.Nullable<T> struct. Think it as something like the following structure.

**Example**: Nullable struct

[Serializable]

public struct Nullable<T> where T : struct

{

public bool HasValue { get; }

public T Value { get; }

// other implementation

}

A nullable of type *int* is the same as an ordinary *int* plus a flag that says whether the *int* has a value or not (is null or not). All the rest is compiler magic that treats "null" as a valid value.

**Example**: HasValue

static void Main(string[] args)

{

Nullable<int> i = null;

if (i.HasValue)

Console.WriteLine(i.Value); // or Console.WriteLine(i)

else

Console.WriteLine("Null");

}

**Output**:

Null

The HasValue returns **true** if the object has been assigned a value; if it has not been assigned any value or has been assigned a null value, it will return **false**.

Accessing the value using NullableType.value will throw a runtime exception if nullable type is null or not assigned any value. For example, i.Value will throw an exception if i is null:

[Text

Description automatically generated](https://www.tutorialsteacher.com/Content/images/csharp/nullabletype-error.png)

Use the GetValueOrDefault() method to get an actual value if it is not null and the default value if it is null. For example:

**Example**: GetValueOrDefault()

static void Main(string[] args)

{

Nullable<int> i = null;

Console.WriteLine(i.GetValueOrDefault());

}

### Shorthand Syntax for Nullable Types

You can use the '?' operator to shorthand the syntax e.g. int?, long? instead of using Nullable<T>.

**Example**: Shorthand syntax for Nullable types

int? i = null;

double? D = null;

### ?? Operator

Use the '??' operator to assign a nullable type to a non-nullable type.

**Example**: ?? operator with Nullable Type

int? i = null;

int j = i ?? 0;

Console.WriteLine(j);

**Output**:

0

In the above example, i is a nullable int and if you assign it to the non-nullable int j then it will throw a runtime exception if i is null. So to mitigate the risk of an exception, we have used the '??' operator to specify that if i is null then assign 0 to j.

### Assignment Rules

A nullable type has the same assignment rules as a value type. It must be assigned a value before using it if nullable types are declared in a function as local variables. If it is a field of any class then it will have a null value by default.

For example, the following nullable of int type is declared and used without assigning any value. The compiler will give **"Use of unassigned local variable 'i'"**error:

[Text

Description automatically generated with medium confidence](https://www.tutorialsteacher.com/Content/images/csharp/unassigned-nullabletype.png)

In the following example, a nullable of int type is a field of the class, so it will not give any error.

**Example**: Nullable type as Class Field

class MyClass

{

public Nullable<int> i;

}

class Program

{

static void Main(string[] args)

{

MyClass mycls = new MyClass();

if(mycls.i == null)

Console.WriteLine("Null");

}

}

## Value Type vs Reference Type

In C#, these data types are categorized based on how they store their value in the memory. C# includes the following categories of data types:

1. Value type
2. Reference type

### Value Type

A data type is a value type if it holds a data value within its own memory space. It means the variables of these data types directly contain values.

Note: All the value types derive from *System.ValueType*, which in-turn, derives from *System.Object*.

For example, consider integer variable int i = 100;

The system stores 100 in the memory space allocated for the variable i. The following image illustrates how 100 is stored at some hypothetical location in the memory (0x239110) for 'i':

[Arrow

Description automatically generated with medium confidence](https://www.tutorialsteacher.com/Content/images/csharp/value-type-memory-allocation.png)

Memory Allocation of Value Type Variable

The following data types are all of value type:

|  |  |  |  |
| --- | --- | --- | --- |
| bool | Byte | char | decimal |
| double | enum | float | int |
| long | sbyte | short | struct |
| uint | ulong | ushort |  |

### Passing Value Type Variables

When you pass a value-type variable from one method to another, the system creates a separate copy of a variable in another method. If value got changed in the one method, it wouldn't affect the variable in another method.

**Example**: Passing Value Type Variables

static void ChangeValue(int x)

{

x = 200;

Console.WriteLine(x);

}

static void Main(string[] args)

{

int i = 100;

Console.WriteLine(i);

ChangeValue(i);

Console.WriteLine(i);

}

**Output**:

100

200

300

In the above example, variable i in the Main() method remains unchanged even after we pass it to the ChangeValue() method and change it's value there.

### Reference Type

Unlike value types, a reference type doesn't store its value directly. Instead, it stores the address where the value is being stored. In other words, a reference type contains a pointer to another memory location that holds the data.

For example, consider the following string variable:

string s = "Hello World!!";

The following image shows how the system allocates the memory for the above string variable.

[Diagram

Description automatically generated](https://www.tutorialsteacher.com/Content/images/csharp/raference-type-memory-allocation.png)

Memory Allocation of Reference Type Variable

As you can see in the above image, the system selects a random location in memory (0x803200) for the variable s. The value of a variable s is 0x600000, which is the memory address of the actual data value. Thus, reference type stores the address of the location where the actual value is stored instead of the value itself.

The followings are reference type data types:

* String
* Arrays (even if their elements are value types)
* Class
* Delegate

### Passing Reference Type Variables

When you pass a reference type variable from one method to another, it doesn't create a new copy; instead, it passes the variable's address. So, If we change the value of a variable in a method, it will also be reflected in the calling method.

**Example**: Passing Reference Type Variable

static void ChangeReferenceType(Student std2)

{

std2.StudentName = "Steve";

}

static void Main(string[] args)

{

Student std1 = new Student();

std1.StudentName = "Bill";

ChangeReferenceType(std1);

Console.WriteLine(std1.StudentName);

}

**Output**:

Steve

In the above example, we pass the Student object std1 to the ChangeReferenceType() method. Here, it actually pass the memory address of std1. Thus, when the ChangeReferenceType() method changes StudentName, it is actually changing StudentName of std1 object, because std1 and std2 are both pointing to the same address in memory.

String is a reference type, but it is immutable. It means once we assigned a value, it cannot be changed. If we change a string value, then the compiler creates a new string object in the memory and point a variable to the new memory location. So, passing a string value to a function will create a new variable in the memory, and any change in the value in the function will not be reflected in the original value, as shown below.

**Example**: Passing String

static void ChangeReferenceType(string name)

{

name = "Steve";

}

static void Main(string[] args)

{

string name = "Bill";

ChangeReferenceType(name);

Console.WriteLine(name);

}

**Output**:

Bill

### Null

The default value of a reference type variable is null when they are not initialized. Null means not referring to any object.

[Diagram

Description automatically generated with medium confidence](https://www.tutorialsteacher.com/Content/images/csharp/null.png)

Null Reference Type

A value type variable cannot be null because it holds value, not a memory address. C# 2.0 introduced nullable types, using which you can assign null to a value type variable or declare a value type variable without assigning a value to it.

## Struct

In C#, struct is the value type data type that represents data structures. It can contain a parameterized constructor, static constructor, constants, fields, methods, properties, indexers, operators, events, and nested types.

struct can be used to hold small data values that do not require inheritance, e.g. coordinate points, key-value pairs, and complex data structure.

### Structure Declaration

A structure is declared using struct keyword. The default modifier is internal for the struct and its members.

The following example declares a structure Coordinate for the graph.

**Example**: Structure

struct Coordinate

{

public int x;

public int y;

}

A struct object can be created with or without the new operator, same as primitive type variables.

**Example**: Create Structure

struct Coordinate

{

public int x;

public int y;

}

Coordinate point = new Coordinate();

Console.WriteLine(point.x); //output: 0

Console.WriteLine(point.y); //output: 0

Above, an object of the Coordinate structure is created using the new keyword. It calls the default parameterless constructor of the struct, which initializes all the members to their default value of the specified data type.

If you declare a variable of struct type without using new keyword, it does not call any constructor, so all the members remain unassigned. Therefore, you must assign values to each member before accessing them, otherwise, it will give a compile-time error.

**Example**: Create Structure Without new Keyword

struct Coordinate

{

public int x;

public int y;

}

Coordinate point;

Console.Write(point.x); // Compile time error

point.x = 10;

point.y = 20;

Console.Write(point.x); //output: 10

Console.Write(point.y); //output: 20

### Constructors in Structure

A struct cannot contain a parameterless constructor. It can only contain parameterized constructors or a static constructor.

**Example**: Parameterized Constructor in Struct

struct Coordinate

{

public int x;

public int y;

public Coordinate(int x, int y)

{

this.x = x;

this.y = y;

}

}

Coordinate point = new Coordinate(10, 20);

Console.WriteLine(point.x); //output: 10

Console.WriteLine(point.y); //output: 20

You must include all the members of the struct in the parameterized constructor and assign parameters to members; otherwise C# compiler will give a compile-time error if any member remains unassigned.

### Methods and Properties in Structure

A struct can contain properties, auto-implemented properties, methods, etc., same as classes.

**Example**: Methods and Properties in Struct

struct Coordinate

{

public int x { get; set; }

public int y { get; set; }

public void SetOrigin()

{

this.x = 0;

this.y = 0;

}

}

Coordinate point = Coordinate();

point.SetOrigin();

Console.WriteLine(point.x); //output: 0

Console.WriteLine(point.y); //output: 0

The following struct includes the static method.

**Example**: Static Constructor in Struct

struct Coordinate

{

public int x;

public int y;

public Coordinate(int x, int y)

{

this.x = x;

this.y = y;

}

public static Coordinate GetOrigin()

{

return new Coordinate();

}

}

Coordinate point = Coordinate.GetOrigin();

Console.WriteLine(point.x); //output: 0

Console.WriteLine(point.y); //output: 0

# C# - Conversion – Boxing & Unboxing, User Defined

Boxing and unboxing is an important concept in **C#**. C# Type System contains **three data types**: **Value Types (int, char, etc)**, **Reference Types (object)** and **Pointer Types**. Basically it convert a Value Type to a Reference Type, and vice versa. Boxing and Unboxing enables a unified view of the type system in which a value of any type can be treated as an object.

## Boxing

* The process of Converting a **Value Type (char, int etc.) to a Reference Type(object)** is called **Boxing**.
* Boxing is implicit conversion process in which object type (super type) is used.
* The Value type is always stored in Stack. The Referenced Type is stored in Heap.

**Example**

int num = 23; // 23 will assigned to num

Object Obj = num; // Boxing

**Description:** First declare a value type variable (num), which is integer type and assigned it with value 23. Now create a references object type (obj) and applied Explicit operation which results in num value type to be copied and stored in object reference type obj

Let’s understand **Boxing** with a C# programming code:

|  |
| --- |
| // C# implementation to demonstrate  // the Boxing  **using** System;  **class** GFG {      // Main Method  **static** **public** **void** Main()      {          // assigned int value          // 2020 to num  **int** num = 2020;            // boxing  **object** obj = num;            // value of num to be change          num = 100;            System.Console.WriteLine          ("Value - type value of num is : {0}", num);          System.Console.WriteLine          ("Object - type value of obj is : {0}", obj);      }  } |

**Output:**

Value - type value of num is : 100

Object - type value of obj is : 2020

## Unboxing

* The process of convertingreference type**into the**value type is known as **Unboxing**.
* It is explicit conversion process.

**Example:**

int num = 23; // value type is int and assigned value 23

Object Obj = num; // Boxing

int i = (int)Obj; // Unboxing

**Description :** Declaration a value type variable (num), which is integer type and assigned with integer value 23. Now, create a reference object type (obj).The explicit operation for boxing create an value type integer i and applied casting method. Then the referenced type residing on Heap is copy to stack

Let’s understand **Unboxing** with a C# programming code:

// C# implementation to demonstrate

// the Unboxing

**using** System;

**class** GFG {

    // Main Method

**static** **public** **void** Main()

    {

        // assigned int value

        // 23 to num

**int** num = 23;

        // boxing

**object** obj = num;

        // unboxing

**int** i = (**int**)obj;

        // Display result

        Console.WriteLine("Value of ob object is : " + obj);

        Console.WriteLine("Value of i is : " + i);

    }

}

**Output:**

Value of ob object is : 23

Value of i is : 23

## User Defined Conversion

In C# programming, a situation may arise when we have to convert a data type into our own custom type or vice-versa. The custom type may be any class, object, or struct that we created. To perform these types of special conversions, we have to define a special method. The syntax of these special methods is shown below.

**static** **public** **implicit**/**explicit** **operator** ConvertToType(ConvertFromDataType value)

{

    // Conversion implementation and return type of this method must be 'ConvertToType'

**return** ConvertToType;

}

Here, in the above method, '*ConvertFromDataType*' is the data type we are going to change and '*ConvertToType*' will be the converted type. We use implicit or explicit depending on the situation. The return type of this method should be of Converted type. Understand the below example to have more clarity around it.

**using** System;

**namespace** Tutpoint

{

**class** Program

    {

**class** Student

        {

**public** **string** Name { **get**; **set**; }

**public** **int** Roll\_No { **get**; **set**; }

            // This method used to convert int value into 'Student' type

**static** **public** **implicit** **operator** Student(**int** value)

            {

                // Return type of this method should be of 'Student' type

**return** **new** Student { Name = "jainy", Roll\_No = value };

            }

            // This method used to convert 'Student' type value into 'int' type

**static** **public** **explicit** **operator** **int**(Student student)

            {

                // Return type of this method should be of 'int' type

**return** student.Roll\_No;

            }

        }

**static** **void** Main(**string**[] args)

        {

            // Object of Student class is initialised

            Student student = **new** Student();

**int** value = 100;

            // On below line, int value is converted to student type, implicit

            student = value;

            Console.WriteLine("Student name" + student.Name + " Student Roll No." + student.Roll\_No);

            // On below line, value is assigned from student.Roll\_No

            value = student.Roll\_No;

            Console.WriteLine("Roll No. " + value);

            // On below line, student type is converted to int type

            value = (**int**)student;

            Console.WriteLine("Roll No. " + value);

            Console.ReadKey();

        }

    }

}

In this example, we have created a class named '*Student*' which contains two property members '*Name*' and '*Roll\_No*'. We have also written two special conversion methods to convert from type '*Student*' to type '*int*' and vice-versa.

The conversion from "int" to "Student" can be performed directly. As we are converting from smaller to larger data types, so we use implicit on method definition, while the conversion from '*Student*' to '*int*' requires a cast operator as we are converting from larger to smaller type and we use explicit on method definition.

In '*Main*' method, we have directly assigned an int value to Student type without compiling for errors due to the method defined above otherwise, an error will produce as "Cannot implicitly convert type 'int' to 'Tutpoint.Program.Student'". The same happens for conversion from "student" to "int" type.

# C# - Generics

**Generic**is a class which allows the user to define classes and methods with the placeholder. Generics were added to version 2.0 of the C# language. The basic idea behind using Generic is to allow type (Integer, String, … etc and user-defined types) to be a parameter to methods, classes, and interfaces. A primary limitation of collections is the absence of effective type checking. This means that you can put any object in a collection because all classes in the C# programming language extend from the object base class. This compromises type safety and contradicts the basic definition of C# as a type-safe language. In addition, using collections involves a significant performance overhead in the form of implicit and explicit type casting that is required to add or retrieve objects from a collection.  
To address the type safety issue, the **.NET**framework provides generics to create classes, structures, interfaces, and methods that have placeholders for the types they use. Generics are commonly used to create type-safe collections for both reference and value types. The **.NET** framework provides an extensive set of interfaces and classes in the System.Collections.Generic namespace for implementing generic collections.

## **Generic Class**

Generics in C# is its most powerful feature. It allows you to define the type-safe data structures. This out-turn in a remarkable performance boost and high-grade code, because it helps to reuse data processing algorithms without replicating type-specific code. Generics are similar to templates in C++ but are different in implementation and capabilities. Generics introduces the concept of type parameters, because of which it is possible to create methods and classes that defers the framing of data type until the class or method is declared and is instantiated by client code. Generic types perform better than normal system types because they reduce the need for boxing, unboxing, and type casting the variables or objects.

Parameter types are specified in generic class creation.

// C# program to show working of

// user defined Generic classes

**using** System;

// We use < > to specify Parameter type

**public** **class** GFG<T> {

    // private data members

**private** T data;

    // using properties

**public** T value

    {

        // using accessors

**get**

        {

**return** **this**.data;

        }

**set**

        {

**this**.data = value;

        }

    }

}

// Driver class

**class** Test {

    // Main method

**static** **void** Main(**string**[] args)

    {

        // instance of string type

        GFG<**string**> name = **new** GFG<**string**>();

        name.value = "GeeksforGeeks";

        // instance of float type

        GFG<**float**> version = **new** GFG<**float**>();

        version.value = 5.0F;

        // display GeeksforGeeks

        Console.WriteLine(name.value);

        // display 5

        Console.WriteLine(version.value);

    }

}

**Output:**

GeeksforGeeks

5

**Explanation:** The preceding example defines a generic class, GFG, which uses a generic type parameter ‘T’. In the Main() method, two instances of GFG have been created by replacing ‘T’ with ‘string’ and ‘float’ data types. These objects are used to store ‘string’ and ‘float’ values respectively. The GFG class ensures type safety by accepting the required type in its constructor.

**A Generic method with various parameters:** Just as a method can take one argument, generics can take various parameters. One argument can be passed as a familiar type and other as a generic type, as shown below:

**Example**:

// C# program to show multiple

// type parameters in Generics

**using** System;

**public** **class** GFG {

    // Generics method

**public** **void** Display<TypeOfValue>(**string** msg, TypeOfValue value)

    {

        Console.WriteLine("{0}:{1}", msg, value);

    }

}

// Driver class

**public** **class** Example {

    // Main Method

**public** **static** **int** Main()

    {

        // creating object of class GFG

        GFG p = **new** GFG();

        // calling Generics method

        p.Display<**int**>("Integer", 122);

        p.Display<**char**>("Character", 'H');

        p.Display<**double**>("Decimal", 255.67);

**return** 0;

    }

}

**Output**:

Integer:122

Character:H

Decimal:255.67

## Features of Generics

Generics is a technique that improves your programs in many ways such as:

* It helps you in code reuse, performance and type safety.
* You can create your own generic classes, methods, interfaces and delegates.
* You can create generic collection classes. The .NET framework class library contains many new generic collection classes in System.Collections.Generic namespace.
* You can get information on the types used in generic data type at run-time.

## Advantages of Generics

* **Reusability:** You can use a single generic type definition for multiple purposes in the same code without any alterations. For example, you can create a generic method to add two numbers. This method can be used to add two integers as well as two floats without any modification in the code.
* **Type Safety:** Generic data types provide better type safety, especially in the case of collections. When using generics, you need to define the type of objects to be passed to a collection. This helps the compiler to ensure that only those object types that are defined in the definition can be passed to the collection.
* **Performance:** Generic types provide better performance as compared to normal system types because they reduce the need for boxing, unboxing, and typecasting of variables or objects.

# C# - Logging (Serilog)

We shall be leveraging **DI** (Dependency Injection) framework to inject the Serilog logger object into the Console application pipeline.

Unlike ASP.NET Core the Console application doesn’t have dependency injection by default.

In ASP.NET WebAPI it was easy to configure and use **ILogger**through DI.

**ILogger**interface works very nicely with the .NET Core ecosystem and today in this post we will learn how to enable logging in to a .NET Core Console application.

*File/Rolling file logging providers are still not available through the .NET Core Framework. We need to rely on external solutions for high-end logging requirements like file or database logging.*

*Microsoft recommends using a third-party logger framework like a****Serilog****or****NLog****and many other frameworks for other high-end logging requirements like****Database or File/Rolling File logging****.*

## Minimum level

Serilog implements the common concept of a 'minimum level' for log event processing.

Log.Logger = new LoggerConfiguration()

.MinimumLevel.Debug()

.WriteTo.Console()

.CreateLogger();

The MinimumLevel configuration object provides for one of the log event levels to be specified as the minimum. In the example above, log events with level Debug and higher will be processed and ultimately written to the console.

**Default Level** - if no MinimumLevel is specified, then Information level events and higher will be processed.

| **Level** | **Usage** |
| --- | --- |
| Verbose | Verbose is the noisiest level, rarely (if ever) enabled for a production app. |
| Debug | Debug is used for internal system events that are not necessarily observable from the outside, but useful when determining how something happened. |
| Information | Information events describe things happening in the system that correspond to its responsibilities and functions. Generally, these are the observable actions the system can perform. |
| Warning | When service is degraded, endangered, or may be behaving outside of its expected parameters, Warning level events are used. |
| Error | When functionality is unavailable or expectations broken, an Error event is used. |
| Fatal | The most critical level, Fatal events demand immediate attention. |

## Getting Started

### Let’s create a .NET Core Console application

Here I have used the .NET Core 3.1 or .NET 5 Console application, but the below technique will work in any lower version also.

Update the **Main()**method as below,

**static** **void** Main(**string**[] args)

        {

**var** services = **new** ServiceCollection();

            ConfigureServices(services);

**using** (ServiceProvider serviceProvider = services.BuildServiceProvider())

            {

                MyApplication app = serviceProvider.GetService<MyApplication>();

                app.Run();

            }

        }

Above is a basic DI Container code which will help us in doing DI (Dependency Injection) of logger or business objects.

Please add below NuGet packages explicitly to your application.

* Microsoft.Extensions.DependencyInjection
* Microsoft.Extensions.Logging
* Serilog.Extensions.Logging

### Add basic File Logging

Install-Package **Serilog.Sinks.File** -Version 4.1.0

**private** **static** **void** ConfigureServices(ServiceCollection services)

       {

           services.AddTransient<MyApplication>()

                   .AddScoped<IBusinessLayer, CBusinessLayer>()

                   .AddSingleton<IDataAccessLayer, CDataAccessLayer>();

**var** serilogLogger = **new** LoggerConfiguration()

           .WriteTo.File("TheCodeBuzz.txt")

           .CreateLogger();

           services.AddLogging(builder =>

           {

               builder.SetMinimumLevel(LogLevel.Information);

               builder.AddSerilog(logger: serilogLogger, dispose: **true**);

           });

In the above code, we added logging and custom Startup MyApplication to services collection and built the ***ServiceProvider***for the required services which include **Serilog**object and other business objects.

### Add Rolling File Logging

If you need to enable rolling file logging please install below NuGet package.

Install-Package **Serilog.Sinks.RollingFile** -Version 3.3.0

Updated ConfigureServices method as below,

**private** **static** **void** ConfigureServices(ServiceCollection services)

        {

            services.AddTransient<MyApplication>()

                    .AddScoped<IBusinessLayer, CBusinessLayer>()

                    .AddSingleton<IDataAccessLayer, CDataAccessLayer>();

**var** serilogLogger = **new** LoggerConfiguration()

            .WriteTo.RollingFile("Rolling-TheCodeBuzz.txt")

            .CreateLogger();

            services.AddLogging(builder =>

            {

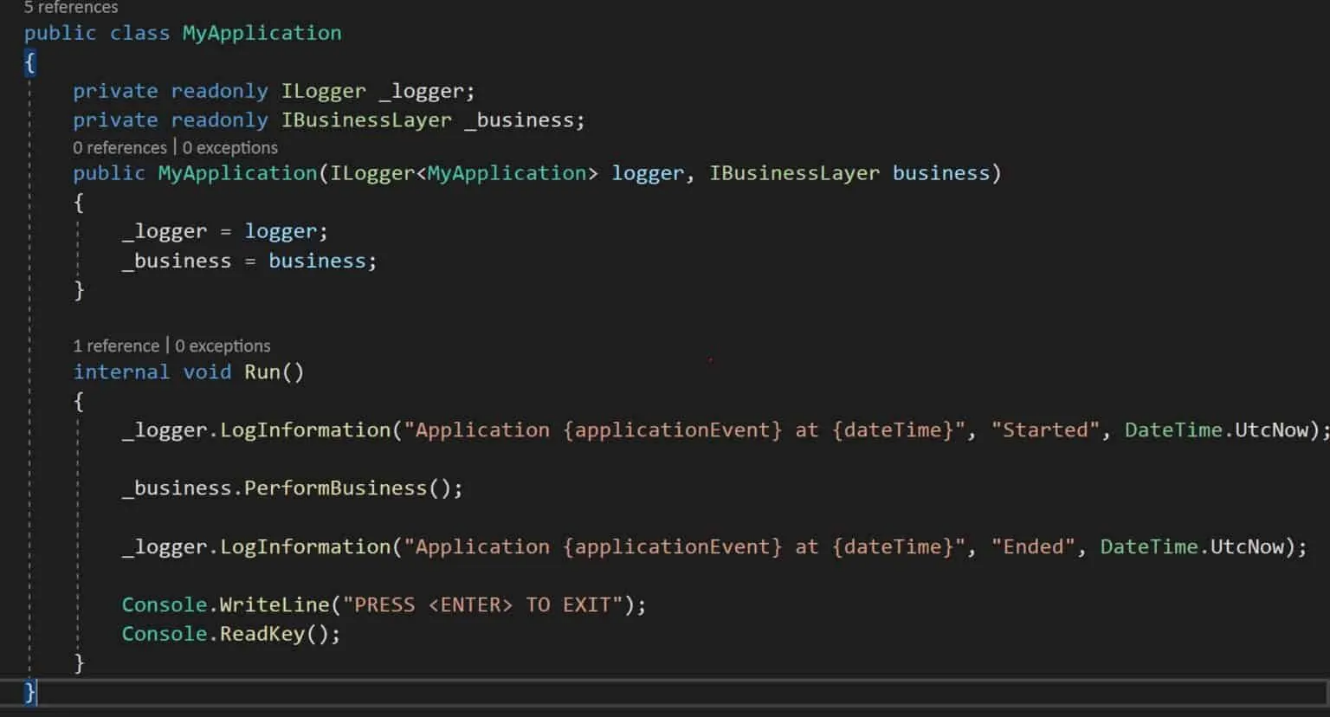
                builder.SetMinimumLevel(LogLevel.Information);

                builder.AddSerilog(logger: serilogLogger, dispose: **true**);

            });

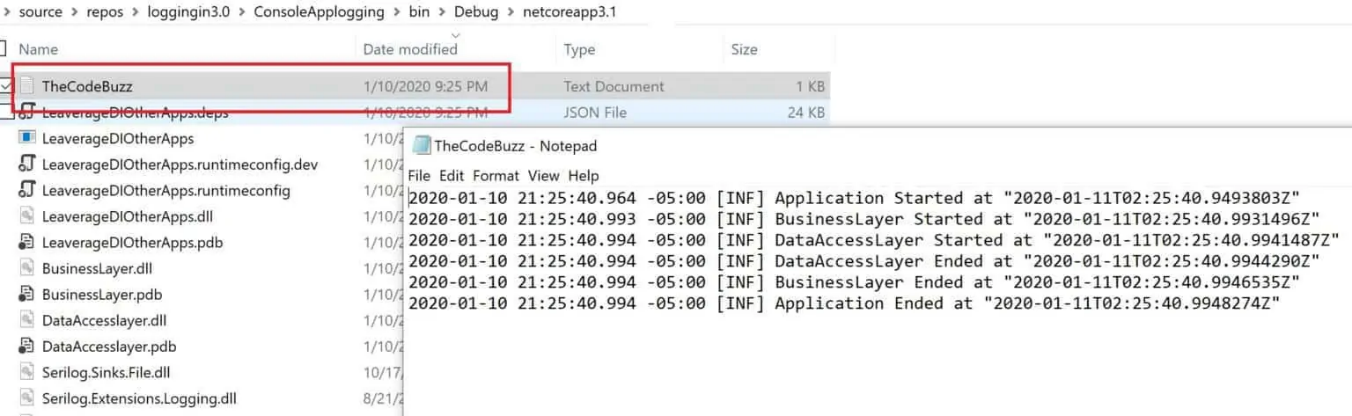
        }

Implementation for class ‘MyApplication’ as below,



Let’s run the application and check log file generated in a file.

Log files shall be generated in the app root folder itself by default (if the path is not specified).



## Another Example

We are going to build a sample application which will mimic connecting to a database through dependency injection as well as outputting logs.

We will start by creating our application, inside our terminal

dotnet new console -n "SampleApp"

Once the application has been create, open the application in Visual Studio Code and let us build and the application to make sure everything is working.

dotnet build

dotnet run

The next step is installing the packages that we need.

**.NET 6**:

dotnet add <CSPROJ PATH> package Microsoft.Extensions.Hosting

dotnet add <CSPROJ PATH> package Serilog.Extensions.Hosting

dotnet add <CSPROJ PATH> package Serilog.Settings.Configuration

dotnet add <CSPROJ PATH> package Serilog.Sinks.Console

**Before .NET 6**:

dotnet add package Microsoft.Extensions.Hosting

dotnet add package Serilog.Extensions.Hosting

dotnet add package Serilog.Settings.Configuration

dotnet add package Serilog.Sinks.Console

The next step will be adding our appsettings.json, to do that in root directory of our application right-click select New File. Name the file appsettings.json

Inside the appsettings we are going to add all of the configuration that we need to setup serilog as well as the connectionString to mimic a database connection

{

"Serilog" : {

"MinimalLevel": {

"Default": "Information",

"Override": {

"Microsoft": "Information",

"System": "Warning"

}

}

},

"ConnectionStrings": {

"DefaultConnection": "DataSource=app.db;Cache=Shared"

}

}

We will start by implementing the logging mechanism. Inside our Program.cs Add the following code, this code responsibility is reading the appsetting.json and making it available to our application.

static void BuildConfig(IConfigurationBuilder builder)

{

// Check the current directory that the application is running on

// Then once the file 'appsetting.json' is found, we are adding it.

// We add env variables, which can override the configs in appsettings.json

builder.SetBasePath(Directory.GetCurrentDirectory())

.AddJsonFile("appsettings.json", optional: false, reloadOnChange: true)

.AddEnvironmentVariables();

}

Now we need to create another method which will be out startup method for our application, it will responsible to put everything together. We will define Serilog as well our dependency injection mechanism in .Net Core.

static IHost AppStartup()

{

var builder = new ConfigurationBuilder();

BuildConfig(builder);

// Specifying the configuration for serilog

Log.Logger = new LoggerConfiguration() // initiate the logger configuration

.ReadFrom.Configuration(builder.Build()) // connect serilog to our configuration folder

.Enrich.FromLogContext() //Adds more information to our logs from built in Serilog

.WriteTo.Console() // decide where the logs are going to be shown

.CreateLogger(); //initialise the logger

Log.Logger.Information("Application Starting");

var host = Host.CreateDefaultBuilder() // Initialising the Host

.ConfigureServices((context, services) => { // Adding the DI container for configuration

})

.UseSerilog() // Add Serilog

.Build(); // Build the Host

return host;

}

Now let us implement data service which will mimic a database

Let us create a new class called DataService and an interface called IDataService

// Interface

public interface IDataService

{

void Connect();

}

// Class

public class DataService : IDataService

{

private readonly ILogger<DataService> \_log;

private readonly IConfiguration \_config;

public DataService(ILogger<DataService> log, IConfiguration config)

{

\_log = log;

\_config = config;

}

public void Connect()

{

// Connect to the database

var connectionString = \_config.GetValue<string>("ConnectionStrings:DefaultConnection");

\_log.LogInformation("Connection String {cs}", connectionString);

}

}

Now we need to update our AppStartup method in the Program.cs class to inject the DataService

var host = Host.CreateDefaultBuilder() // Initialising the Host

.ConfigureServices((context, services) => { // Adding the DI container for configuration

\*\*services.AddTransient<IDataService, DataService>(); // Add transient mean give me an instance each time it is being requested.\*\*

})

.UseSerilog() // Add Serilog

.Build(); // Build the Host

And finally let us put everything together in our main method

static void Main(string[] args)

{

var host = AppStartup();

var service = ActivatorUtilities.CreateInstance<DataService>(host.Services);

service.Connect();

}

## Serilog Configuration File

Add the following settings to the appsettings.json file:

"Serilog": {

"Using": [ "Serilog.Sinks.Console" ],

"MinimumLevel": {

"Default": "Verbose",

"Override": {

"Microsoft": "Warning",

"Microsoft.AspNetCore": "Warning",

"System": "Error"

}

},

"WriteTo": [

{

"Name": "Async",

"Args": {

"configure": [

{

"Name": "Console",

"Args": {

"formatter": "Serilog.Formatting.Compact.RenderedCompactJsonFormatter, Serilog.Formatting.Compact"

}

}

]

}

}

]

}

The Using section defines the types of Sinks that will be used. A **Sink** is just the destination of the logs. So, just **download the Serilog.Sinks.Console NuGet package** and add that value to the Using array to use the Console as a Sink.

Then, we have the MinimumLevel object: it defines the minimum levels of logs that will be taken into consideration. Here the default value is Verbose, but you'll probably want it to be Warning in your production environment. In this way, all the logs with a level lower than Warning will be ignored.

Lastly, we have the WriteTo section, which defines the exact configurations of the sinks. Notice the Async value: we need this value because writing logs is an asynchronous operation - logs must be printed in real-time. So, **after you've installed the Serilog.Sinks.Async NuGet package**, you must add the Async value to that object. And then you can configure the different Sinks: here I'm adding some simple JSON Formatters to the Console Sink.

## File Sink

Add the following settings to the appsettings.json file:

"Serilog": {

"Using": [ "Serilog.Sinks.Console", "Serilog.Sinks.File" ],

:

},

"WriteTo": [

{

:

},

{

"Name": "Logger",

"Args": {

"configureLogger": {

"WriteTo": [

{

"Name": "File",

"Args": {

"path": "Logs/applog\_.log",

"outputTemplate": "{Timestamp:o} [Thread:{ThreadId}] [{Level:u3}] ({SourceContext}) {Message}{NewLine}{Exception}",

"rollingInterval": "Day",

"retainedFileCountLimit": 7

}

}

]

}

}

}

]

}

## Overriding per sink

Sometimes it is desirable to write detailed logs to one medium, but less detailed logs to another.

Log.Logger = new LoggerConfiguration()

.MinimumLevel.Debug()

.WriteTo.File("log.txt")

.WriteTo.Console(restrictedToMinimumLevel: LogEventLevel.Information)

.CreateLogger();

In this example debug logs will be written to the rolling file, while only Information level logs and higher will be written to the console.

All provided sinks support the restrictedToMinimumLevel configuration parameter.

**Logger vs. sink minimums** - it is important to realize that the logging level can only be raised for sinks, not lowered. So, if the logger's MinimumLevel is set to Information then a sink with Debug as its specified level will still only see Information level events. This is because the logger-level configuration controls which logging statements will result in the creation of events, while the sink-level configuration only filters these. To create a single logger with a more verbose level, use a separate LoggerConfiguration.

## Enrichers

Enrichers are simple components that add, remove or modify the properties attached to a log event. This can be used for the purpose of attaching a thread id to each event, for example.

class ThreadIdEnricher : ILogEventEnricher

{

public void Enrich(LogEvent logEvent, ILogEventPropertyFactory propertyFactory)

{

logEvent.AddPropertyIfAbsent(propertyFactory.CreateProperty(

"ThreadId", Thread.CurrentThread.ManagedThreadId));

}

}

Enrichers are added using the Enrich configuration object.

Log.Logger = new LoggerConfiguration()

.Enrich.With(new ThreadIdEnricher())

.WriteTo.Console(

outputTemplate: "{Timestamp:HH:mm} [{Level}] ({ThreadId}) {Message}{NewLine}{Exception}")

.CreateLogger();

The configuration above shows how a property added by an enricher can be used in output formatting.

If the enriched property value is constant throughout the application run, the shortcut WithProperty method can be used to simplify configuration.

Log.Logger = new LoggerConfiguration()

.Enrich.WithProperty("Version", "1.0.0")

.WriteTo.Console()

.CreateLogger();

Enrichers and the properties they attach are generally more useful with sinks that use structured storage, where the property values can be viewed and filtered.

## Refs

<https://github.com/serilog/serilog/wiki/Configuration-Basics>

<https://www.code4it.dev/blog/serilog-log-on-console>

<https://stackoverflow.com/questions/40880261/configuring-serilog-rollingfile-with-appsettings-json>

<https://ivorywolf.medium.com/ive-used-the-ubiquitous-log4net-for-as-long-as-i-can-remember-4cfdd60f8b65>

<https://betterstack.com/community/guides/logging/net/how-to-start-logging-with-serilog>

# Unit Testing using xUnit

## Create the solution

In this section, a solution is created that contains the source and test projects. The completed solution has the following directory structure:

/unit-testing-using-dotnet-test

unit-testing-using-dotnet-test.sln

/PrimeService

PrimeService.cs

PrimeService.csproj

/PrimeService.Tests

PrimeService\_IsPrimeShould.cs

PrimeServiceTests.csproj

The following instructions provide the steps to create the test solution.

Open a shell window.

* Run the following command:

dotnet new sln -o unit-testing-using-dotnet-test

The [dotnet new sln](https://docs.microsoft.com/en-us/dotnet/core/tools/dotnet-new) command creates a new solution in the unit-testing-using-dotnet-test directory.

* Change directory to the *unit-testing-using-dotnet-test* folder.
* Run the following command:

dotnet new classlib -o PrimeService

The [dotnet new classlib](https://docs.microsoft.com/en-us/dotnet/core/tools/dotnet-new) command creates a new class library project in the PrimeService folder. The new class library will contain the code to be tested.

* Rename *Class1.cs* to *PrimeService.cs*.
* Replace the code in *PrimeService.cs* with the following code:

using System;

namespace Prime.Services

{

public class PrimeService

{

public bool IsPrime(int candidate)

{

throw new NotImplementedException("Not implemented.");

}

}

}

* The preceding code:
  + Throws a NotImplementedException with a message indicating it's not implemented.
  + Is updated later in the tutorial.
* In the *unit-testing-using-dotnet-test* directory, run the following command to add the class library project to the solution:

dotnet sln add ./PrimeService/PrimeService.csproj

* Create the *PrimeService.Tests* project by running the following command:

dotnet new xunit -o PrimeService.Tests

* The preceding command:
  + - Creates the PrimeService.Tests project in the PrimeService.Tests directory. The test project uses xUnit as the test library.
    - Configures the test runner by adding the following <PackageReference />elements to the project file:
      * Microsoft.NET.Test.Sdk
      * xunit
      * xunit.runner.visualstudio
      * coverlet.collector
* Add the test project to the solution file by running the following command:

dotnet sln add ./PrimeService.Tests/PrimeService.Tests.csproj

* Add the PrimeService class library as a dependency to the *PrimeService.Tests* project:

dotnet add ./PrimeService.Tests/PrimeService.Tests.csproj reference ./PrimeService/PrimeService.csproj

### Commands to create the solution

This section summarizes all the commands in the previous section. Skip this section if you've completed the steps in the previous section.

The following commands create the test solution on a windows machine. For macOS and Unix, update the ren command to the OS version of ren to rename a file:

dotnet new sln -o unit-testing-using-dotnet-test

cd unit-testing-using-dotnet-test

dotnet new classlib -o PrimeService

ren .\PrimeService\Class1.cs PrimeService.cs

dotnet sln add ./PrimeService/PrimeService.csproj

dotnet new xunit -o PrimeService.Tests

dotnet add ./PrimeService.Tests/PrimeService.Tests.csproj reference ./PrimeService/PrimeService.csproj

dotnet sln add ./PrimeService.Tests/PrimeService.Tests.csproj

Follow the instructions for "Replace the code in PrimeService.cs with the following code" in the previous section.

## Create a test

A popular approach in test driven development (TDD) is to write a test before implementing the target code. This tutorial uses the TDD approach. The IsPrime method is callable, but not implemented. A test call to IsPrime fails. With TDD, a test is written that is known to fail. The target code is updated to make the test pass. You keep repeating this approach, writing a failing test and then updating the target code to pass.

Update the PrimeService.Tests project:

* Delete PrimeService.Tests/UnitTest1.cs.
* Create a PrimeService.Tests/PrimeService\_IsPrimeShould.cs file.
* Replace the code in PrimeService\_IsPrimeShould.cs with the following code:

using Xunit;

using Prime.Services;

namespace Prime.UnitTests.Services

{

public class PrimeService\_IsPrimeShould

{

[Fact]

public void IsPrime\_InputIs1\_ReturnFalse()

{

var primeService = new PrimeService();

bool result = primeService.IsPrime(1);

Assert.False(result, "1 should not be prime");

}

}

}

The [Fact] attribute declares a test method that's run by the test runner. From the PrimeService.Tests folder, run dotnet test. The dotnet test command builds both projects and runs the tests. The xUnit test runner contains the program entry point to run the tests. dotnet test starts the test runner using the unit test project.

The test fails because IsPrime hasn't been implemented. Using the TDD approach, write only enough code so this test passes. Update IsPrime with the following code:

public bool IsPrime(int candidate)

{

if (candidate == 1)

{

return false;

}

throw new NotImplementedException("Not fully implemented.");

}

Run dotnet test. The test passes.

## Add more tests

Add prime number tests for 0 and -1. You could copy the test created in the preceding step and make copies of the following code to test 0 and -1. But don't do it, as there's a better way.

var primeService = new PrimeService();

bool result = primeService.IsPrime(1);

Assert.False(result, "1 should not be prime");

Copying test code when only a parameter changes results in code duplication and test bloat. The following xUnit attributes enable writing a suite of similar tests:

* [Theory] represents a suite of tests that execute the same code but have different input arguments.
* [InlineData] attribute specifies values for those inputs.

Rather than creating new tests, apply the preceding xUnit attributes to create a single theory. Replace the following code:

[Fact]

public void IsPrime\_InputIs1\_ReturnFalse()

{

var primeService = new PrimeService();

bool result = primeService.IsPrime(1);

Assert.False(result, "1 should not be prime");

}

with the following code:

[Theory]

[InlineData(-1)]

[InlineData(0)]

[InlineData(1)]

public void IsPrime\_ValuesLessThan2\_ReturnFalse(int value)

{

var result = \_primeService.IsPrime(value);

Assert.False(result, $"{value} should not be prime");

}

In the preceding code, [Theory] and [InlineData] enable testing several values less than two. Two is the smallest prime number.

Add the following code after the class declaration and before the [Theory] attribute:

private readonly PrimeService \_primeService;

public PrimeService\_IsPrimeShould()

{

\_primeService = new PrimeService();

}

Run dotnet test, and two of the tests fail. To make all of the tests pass, update the IsPrime method with the following code:

public bool IsPrime(int candidate)

{

if (candidate < 2)

{

return false;

}

throw new NotImplementedException("Not fully implemented.");

}

Following the TDD approach, add more failing tests, then update the target code.

# Unit Test Code Coverage & Reporting

Unit tests help to ensure functionality and provide a means of verification for refactoring efforts. Code coverage is a measurement of the amount of code that is run by unit tests - either lines, branches, or methods. As an example, if you have a simple application with only two conditional branches of code (branch a, and branch b), a unit test that verifies conditional branch a will report branch code coverage of 50%.

This article discusses the usage of code coverage for unit testing with Coverlet and report generation using ReportGenerator. While this article focuses on C# and xUnit as the test framework, both MSTest and NUnit would also work. Coverlet is an open source project on GitHub that provides a cross-platform code coverage framework for C#. Coverlet is part of the .NET foundation. Coverlet collects Cobertura coverage test run data, which is used for report generation.

Additionally, this article details how to use the code coverage information collected from a Coverlet test run to generate a report. The report generation is possible using another open source project on GitHub - ReportGenerator. ReportGenerator converts coverage reports generated by Cobertura among many others, into human-readable reports in various formats.

## System under test

The "system under test" refers to the code that you're writing unit tests against, this could be an object, service, or anything else that exposes testable functionality. For this article, you'll create a class library that will be the system under test, and two corresponding unit test projects.

### Create a class library

From a command prompt in a new directory named UnitTestingCodeCoverage, create a new .NET standard class library using the dotnet new classlib command:

dotnet new classlib -n Numbers

The snippet below defines a simple PrimeService class that provides functionality to check if a number is prime. Copy the snippet below and replace the contents of the Class1.cs file that was automatically created in the Numbers directory. Rename the Class1.cs file to PrimeService.cs.

namespace System.Numbers

{

public class PrimeService

{

public bool IsPrime(int candidate)

{

if (candidate < 2)

{

return false;

}

for (int divisor = 2; divisor <= Math.Sqrt(candidate); ++divisor)

{

if (candidate % divisor == 0)

{

return false;

}

}

return true;

}

}

}

**Tip**

It is worth mentioning that the Numbers class library was intentionally added to the System namespace. This allows for **System.Math** to be accessible without a using System; namespace declaration. For more information, see **namespace (C# Reference)**.

### Create test projects

Create two new **xUnit Test Project (.NET Core)** templates from the same command prompt using the dotnet new xunit command:

dotnet new xunit -n XUnit.Coverlet.Collector

dotnet new xunit -n XUnit.Coverlet.MSBuild

Both of the newly created xUnit test projects need to add a project reference of the Numbers class library. This is so that the test projects have access to the PrimeService for testing. From the command prompt, use the dotnet add command:

dotnet add XUnit.Coverlet.Collector\XUnit.Coverlet.Collector.csproj reference Numbers\Numbers.csproj

dotnet add XUnit.Coverlet.MSBuild\XUnit.Coverlet.MSBuild.csproj reference Numbers\Numbers.csproj

The MSBuild project is named appropriately, as it will depend on the coverlet.msbuild NuGet package. Add this package dependency by running the dotnet add package command:

cd XUnit.Coverlet.MSBuild && dotnet add package coverlet.msbuild && cd ..

The previous command changed directories effectively scoping to the MSBuild test project, then added the NuGet package. When that was done, it then changed directories, stepping up one level.

Open both of the UnitTest1.cs files, and replace their contents with the following snippet. Rename the UnitTest1.cs files to PrimeServiceTests.cs.

using System.Numbers;

using Xunit;

namespace XUnit.Coverlet

{

public class PrimeServiceTests

{

readonly PrimeService \_primeService;

public PrimeServiceTests() => \_primeService = new PrimeService();

[

Theory,

InlineData(-1), InlineData(0), InlineData(1)

]

public void IsPrime\_ValuesLessThan2\_ReturnFalse(int value) =>

Assert.False(\_primeService.IsPrime(value), $"{value} should not be prime");

[

Theory,

InlineData(2), InlineData(3), InlineData(5), InlineData(7)

]

public void IsPrime\_PrimesLessThan10\_ReturnTrue(int value) =>

Assert.True(\_primeService.IsPrime(value), $"{value} should be prime");

[

Theory,

InlineData(4), InlineData(6), InlineData(8), InlineData(9)

]

public void IsPrime\_NonPrimesLessThan10\_ReturnFalse(int value) =>

Assert.False(\_primeService.IsPrime(value), $"{value} should not be prime");

}

}

### Create a solution

From the command prompt, create a new solution to encapsulate the class library and the two test projects. Using the dotnet sln command:

dotnet new sln -n XUnit.Coverage

This will create a new solution file name XUnit.Coverage in the UnitTestingCodeCoverage directory. Add the projects to the root of the solution.

**For Windows**:

dotnet sln XUnit.Coverage.sln add (ls \*\*/\*.csproj) --in-root

**For Linux**:

dotnet sln XUnit.Coverage.sln add \*\*/\*.csproj --in-root

Build the solution using the dotnet build command:

dotnet build

If the build is successful, you've created the three projects, appropriately referenced projects and packages, and updated the source code correctly. Well done!

## Code coverage tooling

There are two types of code coverage tools:

* **DataCollectors:** DataCollectors monitor test execution and collect information about test runs. They report the collected information in various output formats, such as XML and JSON. For more information, see your first DataCollector.
* **Report generators:** Use data collected from test runs to generate reports, often as styled HTML.

In this section, the focus is on data collector tools. To use Coverlet for code coverage, an existing unit test project must have the appropriate package dependencies, or alternatively rely on .NET global tooling and the corresponding coverlet.console NuGet package.

## Integrate with .NET test

The xUnit test project template already integrates with coverlet.collector by default. From the command prompt, change directories to the XUnit.Coverlet.Collector project, and run the dotnet test command:

cd XUnit.Coverlet.Collector && dotnet test --collect:"XPlat Code Coverage"

**Note**: The "XPlat Code Coverage" argument is a friendly name that corresponds to the data collectors from Coverlet. This name is required but is case insensitive.

As part of the dotnet test run, a resulting coverage.cobertura.xml file is output to the TestResults directory. The XML file contains the results. This is a cross-platform option that relies on the .NET CLI, and it is great for build systems where MSBuild is not available.

**Tip**: As an alternative, you could use the MSBuild package if your build system already makes use of MSBuild. From the command prompt, change directories to the XUnit.Coverlet.MSBuild project, and run the dotnet test command:

dotnet test /p:CollectCoverage=true /p:CoverletOutputFormat=cobertura

The resulting coverage.cobertura.xml file is output.

## Generate reports

Now that you're able to collect data from unit test runs, you can generate reports using ReportGenerator. To install the ReportGenerator NuGet package as a .NET global tool, use the dotnet tool install command:

dotnet tool install -g dotnet-reportgenerator-globaltool

Run the tool and provide the desired options, given the output coverage.cobertura.xml file from the previous test run.

reportgenerator

-reports:"Path\To\TestProject\TestResults\{guid}\coverage.cobertura.xml"

-targetdir:"coveragereport"

-reporttypes:Html

After running this command, an HTML file represents the generated report.

# Mocking using Moq (Dependency Injection)

When writing automated tests it is sometimes useful to isolate the thing(s) being tested from other parts of the system. These ‘other’ parts may still need to be provided, and sometimes the real versions are too hard or cumbersome to use. In these instances “mocked” versions can be created and used.

A mock version of something is an object that can act like the real thing but can be controlled in test code.

Moq (pronounced “mok u” or “mock”) is a library available on NuGet that allows mock objects to be created in test code and it also supports .NET Core.

Moq allows the manipulation of mock objects in many ways, including setting mock methods to return specific values, setting up properties, and matching specific arguments when the thing being tested calls the mock object.

Install Moq:

install-package Moq

For example, the following code shows a class that requires a constructor dependency to be able to operate:

**using** System;

**namespace** Domain

{

**public** **interface** IThingDependency

    {

**string** JoinUpper(**string** a, **string** b);

**int** Meaning { **get**; }

    }

    // "Real" implementation

**public** **class** ThingDependency : IThingDependency

    {

**public** **string** JoinUpper(**string** a, **string** b)

        {

**throw** **new** NotImplementedException();

        }

**public** **int** Meaning => **throw** **new** NotImplementedException();

    }

    // Class we want to test in isolation of ThingDependency

**public** **class** ThingBeingTested

    {

**private** **readonly** IThingDependency \_thingDependency;

**public** **string** FirstName { **get**; **set**; }

**public** **string** LastName { **get**; **set**; }

**public** ThingBeingTested(IThingDependency thingDependency)

        {

            \_thingDependency = thingDependency;

        }

**public** **string** X()

        {

**var** fullName = \_thingDependency.JoinUpper(FirstName, LastName);

**return** $"{fullName} = {\_thingDependency.Meaning}";

        }

    }

}

Without a mock object, to write a test we could use the real ThingDependency:

[Fact]

**public** **void** TestUsingRealDependency()

{

**var** sut = **new** ThingBeingTested(**new** ThingDependency());

    // test code

}

To isolate the ThingBeingTested from the rest of the system, Moq can create a mock version of an IThingDependency:

[Fact]

**public** **void** TestUsingMockDependency()

{

    // create mock version

**var** mockDependency = **new** Mock<IThingDependency>();

    // set up mock version's method

    mockDependency.Setup(x => x.JoinUpper(It.IsAny<**string**>(), It.IsAny<**string**>()))

                  .Returns("A B");

    // set up mock version's property

    mockDependency.Setup(x => x.Meaning)

                  .Returns(42);

    // create thing being tested with a mock dependency

**var** sut = **new** ThingBeingTested(mockDependency.Object);

**var** result = sut.X();

    Assert.Equal("A B = 42", result);

}

In the preceding code, the Setup() method is used to tell the mock how to behave when it is called by the ThingBeingTested.

Moq can also be used to test the correct interactions are occurring between the ThingBeingTested and the IThingDependency:

[Fact]

**public** **void** TestUsingMockDependencyUsingInteractionVerification()

{

    // create mock version

**var** mockDependency = **new** Mock<IThingDependency>();

    // create thing being tested with a mock dependency

**var** sut = **new** ThingBeingTested(mockDependency.Object)

    {

        FirstName = "Sarah",

        LastName = "Smith"

    };

    sut.X();

    // Assert that the JoinUpper method was called with Sarah Smith

    mockDependency.Verify(x => x.JoinUpper("Sarah", "Smith"), Times.Once);

    // Assert that the Meaning property was accessed once

    mockDependency.Verify(x => x.Meaning, Times.Once);

}

In the preceding code, the Verify method is used to check that the mock JoinUpper method is being called exactly once with the values “Sarah” and “Smith”. The test code is also expecting the method to be called exactly once.

Moq can be used to test in isolation other parts of applications such as ASP.NET Core MVC controllers, where the controller requires a dependency (such as an IFooRepository):

[Fact]

**public** **void** ContollerTest()

{

**var** mockDependency = **new** Mock<IFooRepository>();

**var** sut = **new** HomeController(mockDependency.Object);

    // test code

}

# ADO.NET

ADO.NET provides consistent access to data sources such as SQL Server and XML, and to data sources exposed through OLE DB and ODBC. Data-sharing consumer applications can use ADO.NET to connect to these data sources and retrieve, handle, and update the data that they contain.

ADO.NET separates data access from data manipulation into discrete components that can be used separately or in tandem. ADO.NET includes .NET Framework data providers for connecting to a database, executing commands, and retrieving results. Those results are either processed directly, placed in an ADO.NET DataSet object in order to be exposed to the user in an ad hoc manner, combined with data from multiple sources, or passed between tiers. The DataSet object can also be used independently of a .NET Framework data provider to manage data local to the application or sourced from XML.

The ADO.NET classes are found in System.Data.dll, and are integrated with the XML classes found in System.Xml.dll

ADO.NET provides functionality to developers who write managed code similar to the functionality provided to native component object model (COM) developers by ActiveX Data Objects (ADO). We recommend that you use ADO.NET, not ADO, for accessing data in your .NET applications.

ADO.NET provides the most direct method of data access within the .NET Framework.

## ADO.NET Architecture

Data processing has traditionally relied primarily on a connection-based, two-tier model. As data processing increasingly uses multi-tier architectures, programmers are switching to a disconnected approach to provide better scalability for their applications.

The two main components of ADO.NET for accessing and manipulating data are the .NET Framework data providers and the DataSet.

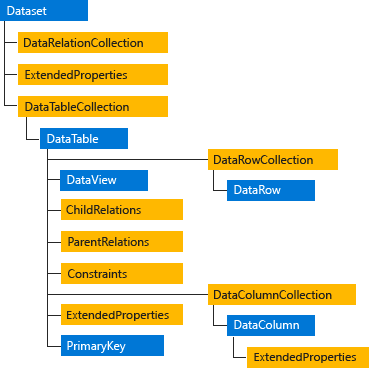
### .NET Framework Data Providers

The .NET Framework Data Providers are components that have been explicitly designed for data manipulation and fast, forward-only, read-only access to data. The Connection object provides connectivity to a data source. The Command object enables access to database commands to return data, modify data, run stored procedures, and send or retrieve parameter information. The DataReader provides a high-performance stream of data from the data source. Finally, the DataAdapter provides the bridge between the DataSet object and the data source. The DataAdapter uses Command objects to execute SQL commands at the data source to both load the DataSet with data and reconcile changes that were made to the data in the DataSet back to the data source.

### The DataSet

The ADO.NET DataSet is explicitly designed for data access independent of any data source. As a result, it can be used with multiple and differing data sources, used with XML data, or used to manage data local to the application. The DataSet contains a collection of one or more DataTable objects consisting of rows and columns of data, and also primary key, foreign key, constraint, and relation information about the data in the DataTable objects.

The following diagram illustrates the relationship between a .NET Framework data provider and a DataSet.

  
ADO.NET architecture

### Choosing a DataReader or a DataSet

When you decide whether your application should use a  or a DataSet, consider the type of functionality that your application requires. Use a DataSet to do the following:

* Cache data locally in your application so that you can manipulate it. If you only need to read the results of a query, the DataReader is the better choice.
* Remote data between tiers or from an XML Web service.
* Interact with data dynamically such as binding to a Windows Forms control or combining and relating data from multiple sources.
* Perform extensive processing on data without requiring an open connection to the data source, which frees the connection to be used by other clients.

If you do not require the functionality provided by the DataSet, you can improve the performance of your application by using the DataReader to return your data in a forward-only, read-only manner. Although the DataAdapter uses the DataReader to fill the contents of a DataSet, by using the DataReader, you can boost performance because you will save memory that would be consumed by the DataSet, and avoid the processing that is required to create and fill the contents of the DataSet.

## LINQ to DataSet

LINQ to DataSet provides query capabilities and compile-time type checking over data cached in a DataSet object. It allows you to write queries in one of the .NET Framework development language, such as C# or Visual Basic.

## LINQ to SQL

Language Integrated Query (*LINQ*) to SQL supports queries against an object model that is mapped to the data structures of a relational database without using an intermediate conceptual model. Each table is represented by a separate class, tightly coupling the object model to the relational database schema. LINQ to SQL translates language-integrated queries in the object model into Transact-SQL and sends them to the database for execution. When the database returns the results, LINQ to SQL translates the results back into objects.

## ADO.NET Entity Framework

The ADO.NET Entity Framework is designed to enable developers to create data access applications by programming against a conceptual application model instead of programming directly against a relational storage schema. The goal is to decrease the amount of code and maintenance required for data-oriented applications.

## WCF Data Services

WCF Data Services is used to deploy data services on the Web or an intranet. The data is structured as entities and relationships according to the specifications of the Entity Data Model. Data deployed on this model is addressable by standard HTTP protocol.

## XML and ADO.NET

ADO.NET leverages the power of XML to provide disconnected access to data. ADO.NET was designed hand-in-hand with the XML classes in the .NET Framework; both are components of a single architecture.

ADO.NET and the XML classes in the .NET Framework converge in the DataSet object. The DataSet can be populated with data from an XML source, whether it is a file or an XML stream. The DataSet can be written as World-Wide Web Consortium (W3C) compliant XML that includes its schema as XML schema definition language (XSD) schema, regardless of the source of the data in the DataSet. Because of the native serialization format of the DataSet is XML, it is an excellent medium for moving data between tiers, making the DataSet an optimal choice for remoting data and schema context to and from an XML Web service.

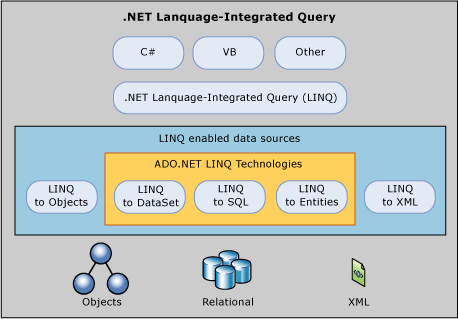
## LINQ

Language-Integrated Query (LINQ) enables developers to form set-based queries in their application code, without having to use a separate query language. You can write LINQ queries against various enumerable data sources (that is, a data source that implements the IEnumerable interface), such as in-memory data structures, XML documents, SQL databases, and DataSet objects. Although these enumerable data sources are implemented in various ways, they all expose the same syntax and language constructs. Because queries can be formed in the programming language itself, you do not have to use another query language that is embedded as string literals that cannot be understood or verified by the compiler. Integrating queries into the programming language also enables Visual Studio programmers to be more productive by providing compile-time type and syntax checking, and IntelliSense. These features reduce the need for query debugging and error fixing.

Transferring data from SQL tables into objects in memory is often tedious and error-prone. The LINQ provider implemented by LINQ to DataSet and LINQ to SQL converts the source data into IEnumerable-based object collections. The programmer always views the data as an IEnumerable collection, both when you query and when you update. Full IntelliSense support is provided for writing queries against those collections.

There are three separate ADO.NET Language-Integrated Query (LINQ) technologies: LINQ to DataSet, LINQ to SQL, and LINQ to Entities. LINQ to DataSet provides richer, optimized querying over the DataSet and LINQ to SQL enables you to directly query SQL Server database schemas, and LINQ to Entities allows you to query an Entity Data Model.

The following diagram provides an overview of how the ADO.NET LINQ technologies relate to high-level programming languages and LINQ-enabled data sources.



### LINQ to DataSet

The DataSet is a key element of the disconnected programming model that ADO.NET is built on, and is widely used. LINQ to DataSet enables developers to build richer query capabilities into DataSet by using the same query formulation mechanism that is available for many other data sources.

### LINQ to SQL

LINQ to SQL is a useful tool for developers who do not require mapping to a conceptual model. By using LINQ to SQL, you can use the LINQ programming model directly over existing database schema. LINQ to SQL enables developers to generate .NET Framework classes that represent data. Rather than mapping to a conceptual data model, these generated classes map directly to database tables, views, stored procedures, and user-defined functions.

With LINQ to SQL, developers can write code directly against the storage schema using the same LINQ programming pattern as in-memory collections and the DataSet, in addition to other data sources such as XML.

### LINQ to Entities

Most applications are currently written on top of relational databases. At some point, these applications will need to interact with the data represented in a relational form. Database schemas are not always ideal for building applications, and the conceptual models of application are not the same as the logical models of databases. The Entity Data Model is a conceptual data model that can be used to model the data of a particular domain so that applications can interact with data as objects.

Through the Entity Data Model, relational data is exposed as objects in the .NET environment. This makes the object layer an ideal target for LINQ support, allowing developers to formulate queries against the database from the language used to build the business logic. This capability is known as LINQ to Entities.

## .NET Framework Data Providers

A .NET Framework data provider is used for connecting to a database, executing commands, and retrieving results. Those results are either processed directly, placed in a DataSet in order to be exposed to the user as needed, combined with data from multiple sources, or remoted between tiers. .NET Framework data providers are lightweight, creating a minimal layer between the data source and code, increasing performance without sacrificing functionality.

The following table lists the data providers that are included in the .NET Framework.

| **.NET FRAMEWORK DATA PROVIDERS** | |
| --- | --- |
| **.NET Framework data provider** | **Description** |
| .NET Framework Data Provider for SQL Server | Provides data access for Microsoft SQL Server. Uses the System.Data.SqlClient namespace. |
| .NET Framework Data Provider for OLE DB | For data sources exposed by using OLE DB. Uses the System.Data.OleDb namespace. |
| .NET Framework Data Provider for ODBC | For data sources exposed by using ODBC. Uses the System.Data.Odbc namespace. |
| .NET Framework Data Provider for Oracle | For Oracle data sources. The .NET Framework Data Provider for Oracle supports Oracle client software version 8.1.7 and later, and uses the System.Data.OracleClient namespace. |
| EntityClient Provider | Provides data access for Entity Data Model (EDM) applications. Uses the System.Data.EntityClient namespace. |
| .NET Framework Data Provider for SQL Server Compact 4.0. | Provides data access for Microsoft SQL Server Compact 4.0. Uses the System.Data.SqlServerCe namespace. |

### Core Objects of .NET Framework Data Providers

The following table outlines the four core objects that make up a .NET Framework data provider.

| **CORE OBJECTS OF .NET FRAMEWORK DATA PROVIDERS** | |
| --- | --- |
| **Object** | **Description** |
| Connection | Establishes a connection to a specific data source. The base class for all Connection objects is the DbConnection class. |
| Command | Executes a command against a data source. Exposes Parameters and can execute in the scope of a Transaction from a Connection. The base class for all Command objects is the DbCommand class. |
| DataReader | Reads a forward-only, read-only stream of data from a data source. The base class for all DataReader objects is the DbDataReader class. |
| DataAdapter | Populates a DataSet and resolves updates with the data source. The base class for all DataAdapter objects is the DbDataAdapter class. |

In addition to the core classes listed in the table earlier in this document, a .NET Framework data provider also contains the classes listed in the following table.

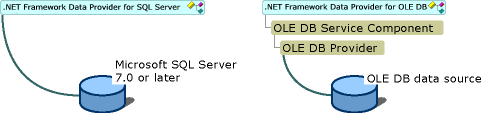
| **CORE OBJECTS OF .NET FRAMEWORK DATA PROVIDERS** | |
| --- | --- |
| **Object** | **Description** |
| Transaction | Enlists commands in transactions at the data source. The base class for all Transaction objects is the DbTransaction class. ADO.NET also provides support for transactions using classes in the System.Transactions namespace. |
| CommandBuilder | A helper object that automatically generates command properties of a DataAdapter or derives parameter information from a stored procedure and populates the Parameters collection of a Command object. The base class for all CommandBuilder objects is the DbCommandBuilder class. |
| ConnectionStringBuilder | A helper object that provides a simple way to create and manage the contents of connection strings used by the Connection objects. The base class for all ConnectionStringBuilder objects is the DbConnectionStringBuilder class. |
| Parameter | Defines input, output, and return value parameters for commands and stored procedures. The base class for all Parameter objects is the DbParameter class. |
| Exception | Returned when an error is encountered at the data source. For an error encountered at the client, .NET Framework data providers throw a .NET Framework exception. The base class for all Exception objects is the DbException class. |
| Error | Exposes the information from a warning or error returned by a data source. |
| ClientPermission | Provided for .NET Framework data provider code access security attributes. The base class for all ClientPermission objects is the DBDataPermission class. |

### .NET Framework Data Provider for SQL Server (SqlClient)

The .NET Framework Data Provider for SQL Server (SqlClient) uses its own protocol to communicate with SQL Server. It is lightweight and performs well because it is optimized to access a SQL Server directly without adding an OLE DB or Open Database Connectivity (ODBC) layer. The following illustration contrasts the .NET Framework Data Provider for SQL Server with the .NET Framework Data Provider for OLE DB. The .NET Framework Data Provider for OLE DB communicates to an OLE DB data source through both the OLE DB Service component, which provides connection pooling and transaction services, and the OLE DB provider for the data source.

**Note**: The .NET Framework Data Provider for ODBC has a similar architecture to the .NET Framework Data Provider for OLE DB; for example, it calls into an ODBC Service Component.

Comparison of the .NET Framework Data Provider for SQL Server and the .NET Framework Data Provider for OLE DB:



The .NET Framework Data Provider for SQL Server classes are located in the System.Data.SqlClient namespace.

The .NET Framework Data Provider for SQL Server supports both local and distributed transactions. For distributed transactions, the .NET Framework Data Provider for SQL Server, by default, automatically enlists in a transaction and obtains transaction details from Windows Component Services or System.Transactions.

The following code example shows how to include the System.Data.SqlClient namespace in your applications.

using System.Data.SqlClient;

## ADO.NET SqlClient Data Provider Examples

The code in this example assumes that you can connect to the Northwind sample database on Microsoft SQL Server. The code creates a SqlCommand to select rows from the Products table, adding a SqlParameter to restrict the results to rows with a UnitPrice greater than the specified parameter value, in this case 5. The SqlConnection is opened inside a using block, which ensures that resources are closed and disposed when the code exits. The code executes the command by using a SqlDataReader, and displays the results in the console window.

### Retrieve Data Using a DataReader

To retrieve data using a **DataReader**, create an instance of the **Command** object, and then create a **DataReader** by calling **Command.ExecuteReader** to retrieve rows from a data source. The **DataReader** provides an unbuffered stream of data that allows procedural logic to efficiently process results from a data source sequentially. The **DataReader** is a good choice when you're retrieving large amounts of data because the data is not cached in memory.

### Closing the DataReader

Always call the **Close** method when you have finished using the **DataReader** object.

If your **Command** contains output parameters or return values, those values are not available until the **DataReader** is closed.

While a **DataReader** is open, the **Connection** is in use exclusively by that **DataReader**. You cannot execute any commands for the **Connection**, including creating another **DataReader**, until the original **DataReader** is closed.

**Note**: Do not call **Close** or **Dispose** on a **Connection**, a **DataReader**, or any other managed object in the **Finalize** method of your class. In a finalizer, only release unmanaged resources that your class owns directly. If your class does not own any unmanaged resources, do not include a **Finalize** method in your class definition.

#### Example

First, add the System.Data.SqlClient NuGet package.

using System;

using System.Data;

using System.Data.SqlClient;

class Program

{

static void Main()

{

string connectionString =

@"Data Source=(local)\SQLEXPRESS;Initial Catalog=Northwind;"

+ "Integrated Security=true";

// Provide the query string with a parameter placeholder.

string queryString =

"SELECT ProductID, UnitPrice, ProductName from dbo.products "

+ "WHERE UnitPrice > @pricePoint "

+ "ORDER BY UnitPrice DESC;";

// Specify the parameter value.

int paramValue = 5;

// Create and open the connection in a using block. This

// ensures that all resources will be closed and disposed

// when the code exits.

using (SqlConnection connection =

new SqlConnection(connectionString))

{

// Create the Command and Parameter objects.

SqlCommand command = new SqlCommand(queryString, connection);

command.Parameters.AddWithValue("@pricePoint", paramValue);

// Open the connection in a try/catch block.

// Create and execute the DataReader, writing the result

// set to the console window.

try

{

connection.Open();

SqlDataReader reader = command.ExecuteReader();

while (reader.Read())

{

Console.WriteLine("\t{0}\t{1}\t{2}",

reader[0], reader[1], reader[2]);

}

reader.Close();

}

catch (Exception ex)

{

Console.WriteLine(ex.Message);

}

Console.ReadLine();

}

}

}

### Executing a Command

Each .NET Framework data provider included with the .NET Framework has its own command object that inherits from DbCommand. The .NET Framework Data Provider for OLE DB includes an OleDbCommand object, the .NET Framework Data Provider for SQL Server includes a SqlCommand object, the .NET Framework Data Provider for ODBC includes an OdbcCommand object, and the .NET Framework Data Provider for Oracle includes an OracleCommand object. Each of these objects exposes methods for executing commands based on the type of command and desired return value, as described in the following table.

| **EXECUTING A COMMAND** | |
| --- | --- |
| Command | Return Value |
| ExecuteReader | Returns a DataReader object. |
| ExecuteScalar | Returns a single scalar value. |
| ExecuteNonQuery | Executes a command that does not return any rows. |
| ExecuteXMLReader | Returns an XmlReader. Available for a SqlCommand object only. |

Each strongly typed command object also supports a CommandType enumeration that specifies how a command string is interpreted, as described in the following table.

| **TABLE 2** | |
| --- | --- |
| **CommandType** | **Description** |
| Text | An SQL command defining the statements to be executed at the data source. |
| StoredProcedure | The name of the stored procedure. You can use the Parameters property of a command to access input and output parameters and return values, regardless of which Execute method is called. When using ExecuteReader, return values and output parameters will not be accessible until the DataReader is closed. |
| TableDirect | The name of a table. |

#### Example

The following code example demonstrates how to create a SqlCommand object to execute a stored procedure by setting its properties. A SqlParameter object is used to specify the input parameter to the stored procedure. The command is executed using the ExecuteReader method, and the output from the SqlDataReader is displayed in the console window.

static void GetSalesByCategory(string connectionString,

string categoryName)

{

using (SqlConnection connection = new SqlConnection(connectionString))

{

// Create the command and set its properties.

SqlCommand command = new SqlCommand();

command.Connection = connection;

command.CommandText = "SalesByCategory";

command.CommandType = CommandType.StoredProcedure;

// Add the input parameter and set its properties.

SqlParameter parameter = new SqlParameter();

parameter.ParameterName = "@CategoryName";

parameter.SqlDbType = SqlDbType.NVarChar;

parameter.Direction = ParameterDirection.Input;

parameter.Value = categoryName;

// Add the parameter to the Parameters collection.

command.Parameters.Add(parameter);

// Open the connection and execute the reader.

connection.Open();

using (SqlDataReader reader = command.ExecuteReader())

{

if (reader.HasRows)

{

while (reader.Read())

{

Console.WriteLine("{0}: {1:C}", reader[0], reader[1]);

}

}

else

{

Console.WriteLine("No rows found.");

}

reader.Close();

}

}

}

## Getting a Single Value from a Table

You may need to return database information that is simply a single value rather than in the form of a table or data stream. For example, you may want to return the result of an aggregate function such as COUNT(\*), SUM(Price), or AVG(Quantity). The **Command** object provides the capability to return single values using the **ExecuteScalar** method. The **ExecuteScalar** method returns, as a scalar value, the value of the first column of the first row of the result set.

The following code example inserts a new value in the database using a SqlCommand. The ExecuteScalar method is used to return the identity column value for the inserted record.

static public int AddProductCategory(string connString, string newName)

{

Console.WriteLine("Getting ID of the new record just created...");

Int32 newProdID = 0;

string sql =

"INSERT INTO Categories(CategoryName, Description) VALUES (@Name, @Desc); "

+ "SELECT CAST(scope\_identity() AS int)";

using (SqlConnection conn = new SqlConnection(connString))

{

SqlCommand cmd = new SqlCommand(sql, conn);

cmd.Parameters.Add("@Name", SqlDbType.VarChar);

cmd.Parameters.Add("@Desc", SqlDbType.VarChar);

cmd.Parameters["@name"].Value = newName;

cmd.Parameters["@desc"].Value = newName;

try

{

conn.Open();

newProdID = (Int32)cmd.ExecuteScalar();

}

catch (Exception ex)

{

Console.WriteLine(ex.Message);

}

}

Console.WriteLine("Done getting ID of the new record just created...");

return (int)newProdID;

}

## Updating Data

SQL statements that modify data (such as INSERT, UPDATE, or DELETE) do not return rows. Similarly, many stored procedures perform an action but do not return rows. To execute commands that do not return rows, create a **Command** object with the appropriate SQL command and a **Connection**, including any required **Parameters**. Execute the command with the **ExecuteNonQuery** method of the **Command** object.

The **ExecuteNonQuery** method returns an integer that represents the number of rows affected by the statement or stored procedure that was executed. If multiple statements are executed, the value returned is the sum of the records affected by all of the statements executed.

### Example

The following code example executes an INSERT statement to insert a record into a database using **ExecuteNonQuery**.

// Assumes connection is a valid SqlConnection.

connection.Open();

string queryString = "INSERT INTO Customers " +

"(CustomerID, CompanyName) Values('NWIND', 'Northwind Traders')";

SqlCommand command = new SqlCommand(queryString, connection);

Int32 recordsAffected = command.ExecuteNonQuery();

The following code example executes the stored procedure created by the sample code in Performing Catalog Operations. No rows are returned by the stored procedure, so the **ExecuteNonQuery** method is used, but the stored procedure does receive an input parameter and returns an output parameter and a return value.

For the OleDbCommand object, the **ReturnValue** parameter must be added to the **Parameters** collection first.

// Assumes connection is a valid SqlConnection.

SqlCommand command = new SqlCommand("InsertCategory" , connection);

command.CommandType = CommandType.StoredProcedure;

SqlParameter parameter = command.Parameters.Add(

"@RowCount", SqlDbType.Int);

parameter.Direction = ParameterDirection.ReturnValue;

parameter = command.Parameters.Add(

"@CategoryName", SqlDbType.NChar, 15);

parameter = command.Parameters.Add("@Identity", SqlDbType.Int);

parameter.Direction = ParameterDirection.Output;

command.Parameters["@CategoryName"].Value = "New Category";

command.ExecuteNonQuery();

Int32 categoryID = (Int32) command.Parameters["@Identity"].Value;

Int32 rowCount = (Int32) command.Parameters["@RowCount"].Value;

## DataSet and DataAdapter

You can use the ADO.NET **DataReader** to retrieve a read-only, forward-only stream of data from a database. Results are returned as the query executes, and are stored in the network buffer on the client until you request them using the **Read** method of the **DataReader**. Using the **DataReader** can increase application performance both by retrieving data as soon as it is available, and (by default) storing only one row at a time in memory, reducing system overhead.

A DataAdapter is used to retrieve data from a data source and populate tables within a DataSet. The DataAdapter also resolves changes made to the DataSet back to the data source. The DataAdapter uses the Connection object of the .NET Framework data provider to connect to a data source, and it uses Command objects to retrieve data from and resolve changes to the data source.

Each .NET Framework data provider included with the .NET Framework has a DbDataReader and a DbDataAdapter object: the .NET Framework Data Provider for OLE DB includes an OleDbDataReader and an OleDbDataAdapter object, the .NET Framework Data Provider for SQL Server includes a SqlDataReader and a SqlDataAdapter object, the .NET Framework Data Provider for ODBC includes an OdbcDataReader and an OdbcDataAdapter object, and the .NET Framework Data Provider for Oracle includes an OracleDataReader and an OracleDataAdapter object.

The ADO.NET DataSet is a memory-resident representation of data that provides a consistent relational programming model independent of the data source. The DataSet represents a complete set of data that includes tables, constraints, and relationships among the tables. Because the DataSet is independent of the data source, a DataSet can include data local to the application, and data from multiple data sources. Interaction with existing data sources is controlled through the DataAdapter.

The SelectCommand property of the DataAdapter is a Command object that retrieves data from the data source. The InsertCommand, UpdateCommand, and DeleteCommand properties of the DataAdapter are Command objects that manage updates to the data in the data source according to modifications made to the data in the DataSet.

The Fill method of the DataAdapter is used to populate a DataSet with the results of the SelectCommand of the DataAdapter. Fill takes as its arguments a DataSet to be populated, and a DataTable object, or the name of the DataTable to be filled with the rows returned from the SelectCommand.

**Note**: Using the DataAdapter to retrieve all of a table takes time, especially if there are many rows in the table. This is because accessing the database, locating and processing the data, and then transferring the data to the client is time-consuming. Pulling all of the table to the client also locks all of the rows on the server. To improve performance, you can use the WHERE clause to greatly reduce the number of rows returned to the client. You can also reduce the amount of data returned to the client by only explicitly listing required columns in the SELECT statement. Another good workaround is to retrieve the rows in batches (such as several hundred rows at a time) and only retrieve the next batch when the client is finished with the current batch.

The Fill method uses the DataReader object implicitly to return the column names and types that are used to create the tables in the DataSet, and the data to populate the rows of the tables in the DataSet. Tables and columns are only created if they do not already exist; otherwise Fill uses the existing DataSet schema. Column types are created as .NET Framework types according to the tables in Data Type Mappings in ADO.NET. Primary keys are not created unless they exist in the data source and DataAdapter**.**MissingSchemaAction is set to MissingSchemaAction**.**AddWithKey. If Fill finds that a primary key exists for a table, it will overwrite data in the DataSet with data from the data source for rows where the primary key column values match those of the row returned from the data source. If no primary key is found, the data is appended to the tables in the DataSet. Fill uses any mappings that may exist when you populate the DataSet.

**Note**: If the SelectCommand returns the results of an OUTER JOIN, the DataAdapter does not set a PrimaryKey value for the resulting DataTable. You must define the PrimaryKey yourself to make sure that duplicate rows are resolved correctly..

The following code example creates an instance of a SqlDataAdapter that uses a SqlConnection to the Microsoft SQL Server Northwind database and populates a DataTable in a DataSet with the list of customers. The SQL statement and SqlConnection arguments passed to the SqlDataAdapter constructor are used to create the SelectCommand property of the SqlDataAdapter.

### Example

// Assumes that connection is a valid SqlConnection object.

string queryString =

"SELECT CustomerID, CompanyName FROM dbo.Customers";

SqlDataAdapter adapter = new SqlDataAdapter(queryString, connString);

DataSet customers = new DataSet();

adapter.Fill(customers, "Customers");

**Note**: The code shown in this example does not explicitly open and close the Connection. The Fill method implicitly opens the Connection that the DataAdapter is using if it finds that the connection is not already open. If Fill opened the connection, it also closes the connection when Fill is finished. This can simplify your code when you deal with a single operation such as a Fill or an Update. However, if you are performing multiple operations that require an open connection, you can improve the performance of your application by explicitly calling the Open method of the Connection, performing the operations against the data source, and then calling the Close method of the Connection. You should try to keep connections to the data source open as briefly as possible to free resources for use by other client applications.

### Multiple Result Sets

If the DataAdapter encounters multiple result sets, it creates multiple tables in the DataSet. The tables are given an incremental default name of TableN, starting with "Table" for Table0. If a table name is passed as an argument to the Fill method, the tables are given an incremental default name of TableNameN, starting with "TableName" for TableName0.

#### Populating a DataSet from Multiple DataAdapters

Any number of DataAdapter objects can be used with a DataSet. Each DataAdapter can be used to fill one or more DataTable objects and resolve updates back to the relevant data source. DataRelation and Constraint objects can be added to the DataSet locally, which enables you to relate data from dissimilar data sources. For example, a DataSet can contain data from a Microsoft SQL Server database, an IBM DB2 database exposed through OLE DB, and a data source that streams XML. One or more DataAdapter objects can handle communication to each data source.

#### Example

The following code example populates a list of customers from the Northwind database on Microsoft SQL Server, and a list of orders from the Northwind database stored in Microsoft Access 2000. The filled tables are related with a DataRelation, and the list of customers is then displayed with the orders for that customer

// Assumes that customerConnection is a valid SqlConnection object.

// Assumes that orderConnection is a valid OleDbConnection object.

SqlDataAdapter custAdapter = new SqlDataAdapter(

"SELECT \* FROM dbo.Customers", customerConnection);

OleDbDataAdapter ordAdapter = new OleDbDataAdapter(

"SELECT \* FROM Orders", orderConnection);

DataSet customerOrders = new DataSet();

custAdapter.Fill(customerOrders, "Customers");

ordAdapter.Fill(customerOrders, "Orders");

DataRelation relation = customerOrders.Relations.Add("CustOrders",

customerOrders.Tables["Customers"].Columns["CustomerID"],

customerOrders.Tables["Orders"].Columns["CustomerID"]);

foreach (DataRow pRow in customerOrders.Tables["Customers"].Rows)

{

Console.WriteLine(pRow["CustomerID"]);

foreach (DataRow cRow in pRow.GetChildRows(relation))

Console.WriteLine("\t" + cRow["OrderID"]);

}

## Transactions and Concurrency

A transaction consists of a single command or a group of commands that execute as a package. Transactions allow you to combine multiple operations into a single unit of work. If a failure occurs at one point in the transaction, all of the updates can be rolled back to their pre-transaction state.

A transaction must conform to the ACID properties—atomicity, consistency, isolation, and durability—in order to guarantee data consistency. Most relational database systems, such as Microsoft SQL Server, support transactions by providing locking, logging, and transaction management facilities whenever a client application performs an update, insert, or delete operation.

**Note**: Transactions that involve multiple resources can lower concurrency if locks are held too long. Therefore, keep transactions as short as possible.

If a transaction involves multiple tables in the same database or server, then explicit transactions in stored procedures often perform better. You can create transactions in SQL Server stored procedures by using the Transact-SQL BEGIN TRANSACTION, COMMIT TRANSACTION, and ROLLBACK TRANSACTION statements.

Transactions involving different resource managers, such as a transaction between SQL Server and Oracle, require a distributed transaction.

### Local Transactions

Transactions in ADO.NET are used when you want to bind multiple tasks together so that they execute as a single unit of work. For example, imagine that an application performs two tasks. First, it updates a table with order information. Second, it updates a table that contains inventory information, debiting the items ordered. If either task fails, then both updates are rolled back.

#### Determining the Transaction Type

A transaction is considered to be a local transaction when it is a single-phase transaction and is handled by the database directly. A transaction is considered to be a distributed transaction when it is coordinated by a transaction monitor and uses fail-safe mechanisms (such as two-phase commit) for transaction resolution.

Each of the .NET Framework data providers has its own Transaction object for performing local transactions. If you require a transaction to be performed in a SQL Server database, select a System.Data.SqlClient transaction. For an Oracle transaction, use the System.Data.OracleClient provider. In addition, there is a DbTransaction class that is available for writing provider-independent code that requires transactions.

**Note**: Transactions are most efficient when they are performed on the server. If you are working with a SQL Server database that makes extensive use of explicit transactions, consider writing them as stored procedures using the Transact-SQL BEGIN TRANSACTION statement.

#### Performing a Transaction Using a Single Connection

In ADO.NET, you control transactions with the Connection object. You can initiate a local transaction with the BeginTransaction method. Once you have begun a transaction, you can enlist a command in that transaction with the Transaction property of a Command object. You can then commit or roll back modifications made at the data source based on the success or failure of the components of the transaction.

**Note**: The EnlistDistributedTransaction method should not be used for a local transaction.

The scope of the transaction is limited to the connection. The following example performs an explicit transaction that consists of two separate commands in the try block. The commands execute INSERT statements against the Production.ScrapReason table in the AdventureWorks SQL Server sample database, which are committed if no exceptions are thrown. The code in the catch block rolls back the transaction if an exception is thrown. If the transaction is aborted or the connection is closed before the transaction has completed, it is automatically rolled back.

#### Example

Follow these steps to perform a transaction.

1. Call the BeginTransaction method of the SqlConnection object to mark the start of the transaction. The BeginTransaction method returns a reference to the transaction. This reference is assigned to the SqlCommand objects that are enlisted in the transaction.
2. Assign the Transaction object to the Transaction property of the SqlCommand to be executed. If a command is executed on a connection with an active transaction, and the Transaction object has not been assigned to the Transaction property of the Command object, an exception is thrown.
3. Execute the required commands.
4. Call the Commit method of the SqlTransaction object to complete the transaction, or call the Rollback method to end the transaction. If the connection is closed or disposed before either the Commit or Rollback methods have been executed, the transaction is rolled back.

The following code example demonstrates transactional logic using ADO.NET with Microsoft SQL Server.

using (SqlConnection connection = new SqlConnection(connectionString))

{

connection.Open();

// Start a local transaction.

SqlTransaction sqlTran = connection.BeginTransaction();

// Enlist a command in the current transaction.

SqlCommand command = connection.CreateCommand();

command.Transaction = sqlTran;

try

{

// Execute two separate commands.

command.CommandText =

"INSERT INTO Production.ScrapReason(Name) VALUES('Wrong size')";

command.ExecuteNonQuery();

command.CommandText =

"INSERT INTO Production.ScrapReason(Name) VALUES('Wrong color')";

command.ExecuteNonQuery();

// Commit the transaction.

sqlTran.Commit();

Console.WriteLine("Both records were written to database.");

}

catch (Exception ex)

{

// Handle the exception if the transaction fails to commit.

Console.WriteLine(ex.Message);

try

{

// Attempt to roll back the transaction.

sqlTran.Rollback();

}

catch (Exception exRollback)

{

// Throws an InvalidOperationException if the connection

// is closed or the transaction has already been rolled

// back on the server.

Console.WriteLine(exRollback.Message);

}

}

}

## Stored Procedures

Stored procedures can accept data as input parameters and can return data as output parameters, result sets, or return values. The sample below illustrates how ADO.NET sends and receives input parameters, output parameters, and return values. The example inserts a new record into a table where the primary key column is an identity column in a SQL Server database.

**Note**; If you are using SQL Server stored procedures to edit or delete data using a **SqlDataAdapter**, make sure that you do not use SET NOCOUNT ON in the stored procedure definition. This causes the rows affected count returned to be zero, which the DataAdapter interprets as a concurrency conflict. In this event, a **DBConcurrencyException** will be thrown.

### Example

The sample uses the following stored procedure to insert a new category into the **Northwind** **Categories** table. The stored procedure takes the value in the **CategoryName** column as an input parameter and uses the SCOPE\_IDENTITY() function to retrieve the new value of the identity field, **CategoryID**, and return it in an output parameter. The RETURN statement uses the @@ROWCOUNT function to return the number of rows inserted.

CREATE PROCEDURE dbo.InsertCategory

@CategoryName nvarchar(15),

@Identity int OUT

AS

INSERT INTO Categories (CategoryName) VALUES(@CategoryName)

SET @Identity = SCOPE\_IDENTITY()

RETURN @@ROWCOUNT

The following code example uses the InsertCategory stored procedure shown above as the source for the InsertCommand of the SqlDataAdapter. The @Identity output parameter will be reflected in the DataSet after the record has been inserted into the database when the Update method of the SqlDataAdapter is called. The code also retrieves the return value.

**Note**: When using the **OleDbDataAdapter**, you must specify parameters with a **ParameterDirection** of **ReturnValue** before the other parameters.

using System;

using System.Data;

using System.Data.SqlClient;

class Program

{

static void Main()

{

string connectionString = GetConnectionString();

ReturnIdentity(connectionString);

// Console.ReadLine();

}

private static void ReturnIdentity(string connectionString)

{

using (SqlConnection connection = new SqlConnection(connectionString))

{

// Create a SqlDataAdapter based on a SELECT query.

SqlDataAdapter adapter = new SqlDataAdapter("SELECT CategoryID, CategoryName FROM dbo.Categories", connection);

// Create a SqlCommand to execute the stored procedure.

adapter.InsertCommand = new SqlCommand("InsertCategory", connection);

adapter.InsertCommand.CommandType = CommandType.StoredProcedure;

// Create a parameter for the ReturnValue.

SqlParameter parameter = adapter.InsertCommand.Parameters.Add("@RowCount", SqlDbType.Int);

parameter.Direction = ParameterDirection.ReturnValue;

// Create an input parameter for the CategoryName.

// You do not need to specify direction for input parameters.

adapter.InsertCommand.Parameters.Add("@CategoryName", SqlDbType.NChar, 15, "CategoryName");

// Create an output parameter for the new identity value.

parameter = adapter.InsertCommand.Parameters.Add("@Identity", SqlDbType.Int, 0, "CategoryID");

parameter.Direction = ParameterDirection.Output;

// Create a DataTable and fill it.

DataTable categories = new DataTable();

adapter.Fill(categories);

// Add a new row.

DataRow categoryRow = categories.NewRow();

categoryRow["CategoryName"] = "New Beverages";

categories.Rows.Add(categoryRow);

// Update the database.

adapter.Update(categories);

// Retrieve the ReturnValue.

Int32 rowCount = (Int32)adapter.InsertCommand.Parameters["@RowCount"].Value;

Console.WriteLine("ReturnValue: {0}", rowCount.ToString());

Console.WriteLine("All Rows:");

foreach (DataRow row in categories.Rows)

{

Console.WriteLine(" {0}: {1}", row[0], row[1]);

}

}

}

static private string GetConnectionString()

{

// To avoid storing the connection string in your code,

// you can retrieve it from a configuration file.

return "Data Source=(local);Initial Catalog=Northwind;Integrated Security=true";

}

}

# REST and SOA

## Service Oriented Architecture

Service-oriented architecture (SOA) was an overused term and has meant different things to different people. But as a common denominator, SOA means that you structure your application by decomposing it into multiple services (most commonly as HTTP services) that can be classified as different types like subsystems or tiers.

Docker containers are useful (but not required) for both traditional service-oriented architectures and the more advanced microservices architectures.

Microservices derive from SOA, but SOA is different from microservices architecture. Features like large central brokers, central orchestrators at the organization level, and the Enterprise Service Bus (ESB) are typical in SOA. But in most cases, these are anti-patterns in the microservice community. In fact, some people argue that "The microservice architecture is SOA done right."

SOA just makes it easier for software components over various networks to work with each other.

Web services which are built as per the SOA architecture tend to make web service more independent. The web services themselves can exchange data with each other and because of the underlying principles on which they are created, they don’t need any sort of human interaction and also don’t need any code modifications. It ensures that the web services on a network can interact with each other seamlessly.

## Service-Oriented Architecture (SOA) Principles

There are 9 types of SOA design principles which are mentioned below

**1. Standardized Service Contract**

Services adhere to a service description. A service must have some sort of description which describes what the service is about. This makes it easier for client applications to understand what the service does.

**2. Loose Coupling**

Less dependency on each other. This is one of the main characteristics of web services which just states that there should be as less dependency as possible between the web services and the client invoking the web service. So, if the service functionality changes at any point in time, it should not break the client application or stop it from working.

**3. Service Abstraction**

Services hide the logic they encapsulate from the outside world. The service should not expose how it executes its functionality; it should just tell the client application on what it does and not on how it does it.

**4. Service Reusability**

Logic is divided into services with the intent of maximizing reuse. In any development company re-usability is a big topic because obviously one wouldn’t want to spend time and effort building the same code again and again across multiple applications which require them. Hence, once the code for a web service is written it should have the ability work with various application types.

**5. Service Autonomy**

Services should have control over the logic they encapsulate. The service knows everything on what functionality it offers and hence should also have complete control over the code it contains.

**6. Service Statelessness**

Ideally, services should be stateless. This means that services should not withhold information from one state to the other. This would need to be done from the client application. An example can be an order placed on a shopping site. Now you can have a web service which gives you the price of a particular item. But if the items are added to a shopping cart and the web page navigates to the page where you do the payment, the responsibility of the price of the item to be transferred to the payment page should not be done by the web service. Instead, it needs to be done by the web application.

**7. Service Discoverability**

Services can be discovered (usually in a service registry). We have already seen this in the concept of the UDDI, which performs a registry which can hold information about the web service.

**8. Service Composability**

Services break big problems into little problems. One should never embed all functionality of an application into one single service but instead, break the service down into modules each with a separate business functionality.

**9. Service Interoperability**

Services should use standards that allow diverse subscribers to use the service. In web services, standards as XML and communication over HTTP is used to ensure it conforms to this principle.

## Service-Oriented Architecture (SOA) Terminology

* **Service Consumer**: It finds records in the broker registry using different find services and then binds to the service provider to invoke one of its web services. Which service the service-consumers require, they should take it into the Registry, bind it with several services, and after that work on it. However, they can reach various services if the service gives various services.
* **Service registry**: It is a service provider that transfer service offers to one or more further service providers. It is also known as a service broker and also called it a repository.
* **Service provider**: It generates a web service and produces the information to the service registry or broker — each provider discusses which service to give more attention: security or easy availability.

## Service-Oriented Architecture (SOA) Advantages

* **Service Reusability**: These applications are built from existing services. Thus, services can be re-used to create many other applications.
* **Platform Independent**: The services are platform independent as people can interact with separate applications over a common language.
* **Easy Maintenance**: As services are independent of each other, they can be updated and transformed easily without harming other services.
* **Availability**: These facilities are effortlessly available to anyone on demand.
* **Parallel Development:** This architecture supports the layer-based design; it gives parallel development.
* **Reliability**: These applications are extra secure because it is simple to test short code rather than large codes
* **Scalability**: Services can work on various servers within an environment, this improves scalability.

## **Disadvantages of**Service-Oriented Architecture **(SOA)**

* SOA depends on the implementation of standards. Without standards, communication between applications requires a lot of time and code.
* SOA is not for: applications with a high level of data transfer, applications that do not require the implementation of the request/response type, and applications that have a short life span.
* Increasingly it becomes difficult and expensive to be able to comply with protocols and speak to service.
* It implies knowing the business processes, classifying them, extracting the functions that are common to them, standardizing them, and forming with them layers of services that will be required by any business process.

## Asynchronous Programming (async, await, Task, Thread)

You can avoid performance bottlenecks and enhance the overall responsiveness of your application by using asynchronous programming. However, traditional techniques for writing asynchronous applications can be complicated, making them difficult to write, debug, and maintain.

C# 5 introduced a simplified approach, async programming, that leverages asynchronous support in the .NET Framework 4.5 and higher, .NET Core, and the Windows Runtime. The compiler does the difficult work that the developer used to do, and your application retains a logical structure that resembles synchronous code. As a result, you get all the advantages of asynchronous programming with a fraction of the effort.

### Async improves responsiveness

Asynchrony is essential for activities that are potentially blocking, such as web access. Access to a web resource sometimes is slow or delayed. If such an activity is blocked in a synchronous process, the entire application must wait. In an asynchronous process, the application can continue with other work that doesn't depend on the web resource until the potentially blocking task finishes.

The following table shows typical areas where asynchronous programming improves responsiveness. The listed APIs from .NET and the Windows Runtime contain methods that support async programming.

| **ASYNC IMPROVES RESPONSIVENESS** | | |
| --- | --- | --- |
| **Application area** | **.NET types with async methods** | **Windows Runtime types with async methods** |
| Web access | HttpClient | Windows.Web.Http.HttpClient SyndicationClient |
| Working with files | JsonSerializer StreamReader StreamWriter XmlReader XmlWriter | StorageFile |
| Working with images |  | MediaCapture BitmapEncoder BitmapDecoder |
| WCF programming | Synchronous and Asynchronous Operations |  |

Asynchrony proves especially valuable for applications that access the UI thread because all UI-related activity usually shares one thread. If any process is blocked in a synchronous application, all are blocked. Your application stops responding, and you might conclude that it has failed when instead it's just waiting.

When you use asynchronous methods, the application continues to respond to the UI. You can resize or minimize a window, for example, or you can close the application if you don't want to wait for it to finish.

The async-based approach adds the equivalent of an automatic transmission to the list of options that you can choose from when designing asynchronous operations. That is, you get all the benefits of traditional asynchronous programming but with much less effort from the developer.

### Async methods are easy to write

The async and await keywords in C# are the heart of async programming. By using those two keywords, you can use resources in .NET Framework, .NET Core, or the Windows Runtime to create an asynchronous method almost as easily as you create a synchronous method. Asynchronous methods that you define by using the async keyword are referred to as async methods.

The following example shows an async method. Almost everything in the code should look familiar to you.

public async Task<int> GetUrlContentLengthAsync()

{

var client = new HttpClient();

Task<string> getStringTask =

client.GetStringAsync("https://docs.microsoft.com/dotnet");

DoIndependentWork();

string contents = await getStringTask;

return contents.Length;

}

void DoIndependentWork()

{

Console.WriteLine("Working...");

}

You can learn several practices from the preceding sample. Start with the method signature. It includes the async modifier. The return type is Task<int> (See "Return Types" section for more options). The method name ends in Async. In the body of the method, GetStringAsync returns a Task<string>. That means that when you await the task you'll get a string (contents). Before awaiting the task, you can do work that doesn't rely on the string from GetStringAsync.

Pay close attention to the await operator. It suspends GetUrlContentLengthAsync:

* GetUrlContentLengthAsync can't continue until getStringTask is complete.
* Meanwhile, control returns to the caller of GetUrlContentLengthAsync.
* Control resumes here when getStringTask is complete.
* The await operator then retrieves the string result from getStringTask.

The return statement specifies an integer result. Any methods that are awaiting GetUrlContentLengthAsync retrieve the length value.

If GetUrlContentLengthAsync doesn't have any work that it can do between calling GetStringAsync and awaiting its completion, you can simplify your code by calling and awaiting in the following single statement.

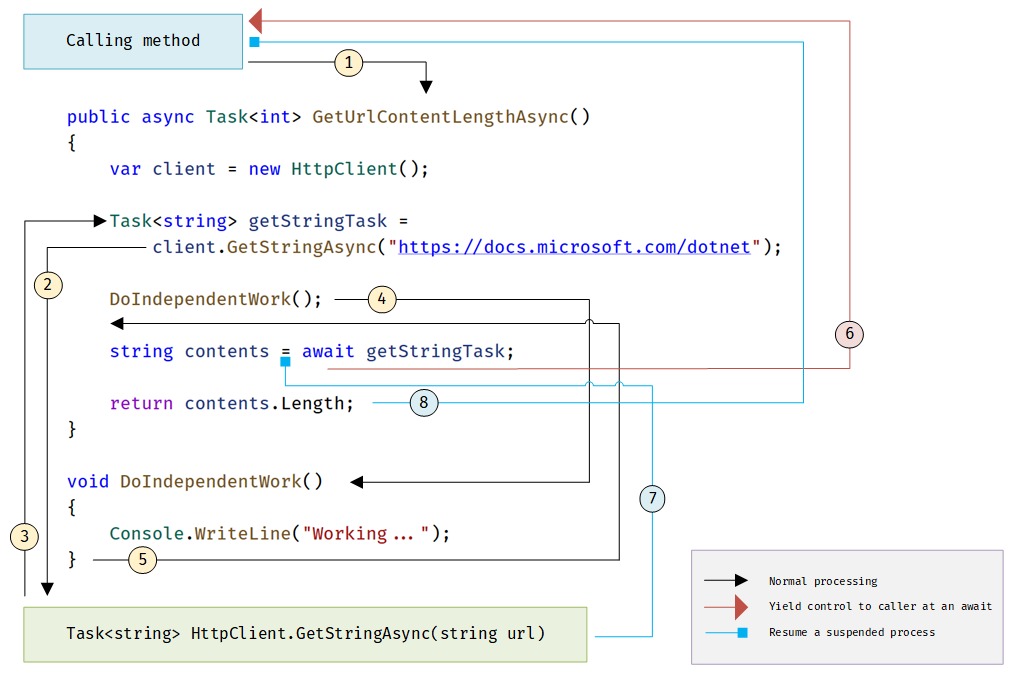
string contents = await client.GetStringAsync("https://docs.microsoft.com/dotnet");

The following characteristics summarize what makes the previous example an async method:

* The method signature includes an async modifier.
* The name of an async method, by convention, ends with an "Async" suffix.
* The return type is one of the following types:
  + Task<TResult> if your method has a return statement in which the operand has type TResult.
  + Task if your method has no return statement or has a return statement with no operand.
  + void if you're writing an async event handler.
  + Any other type that has a GetAwaiter method (starting with C# 7.0).
* The method usually includes at least one await expression, which marks a point where the method can't continue until the awaited asynchronous operation is complete. In the meantime, the method is suspended, and control returns to the method's caller. The next section of this topic illustrates what happens at the suspension point.

### What happens in an async method

The most important thing to understand in asynchronous programming is how the control flow moves from method to method. The following diagram leads you through the process:



The numbers in the diagram correspond to the following steps, initiated when a calling method calls the async method.

1. A calling method calls and awaits the GetUrlContentLengthAsync async method.
2. GetUrlContentLengthAsync creates an HttpClient instance and calls the GetStringAsync asynchronous method to download the contents of a website as a string.
3. Something happens in GetStringAsync that suspends its progress. Perhaps it must wait for a website to download or some other blocking activity. To avoid blocking resources, GetStringAsync yields control to its caller, GetUrlContentLengthAsync.

GetStringAsync returns a Task<TResult>, where TResult is a string, and GetUrlContentLengthAsync assigns the task to the getStringTask variable. The task represents the ongoing process for the call to GetStringAsync, with a commitment to produce an actual string value when the work is complete.

1. Because getStringTask hasn't been awaited yet, GetUrlContentLengthAsync can continue with other work that doesn't depend on the final result from GetStringAsync. That work is represented by a call to the synchronous method DoIndependentWork.
2. DoIndependentWork is a synchronous method that does its work and returns to its caller.
3. GetUrlContentLengthAsync has run out of work that it can do without a result from getStringTask. GetUrlContentLengthAsync next wants to calculate and return the length of the downloaded string, but the method can't calculate that value until the method has the string.

Therefore, GetUrlContentLengthAsync uses an await operator to suspend its progress and to yield control to the method that called GetUrlContentLengthAsync. GetUrlContentLengthAsync returns a Task<int> to the caller. The task represents a promise to produce an integer result that's the length of the downloaded string.

**Note**: If GetStringAsync (and therefore getStringTask) completes before GetUrlContentLengthAsync awaits it, control remains in GetUrlContentLengthAsync. The expense of suspending and then returning to GetUrlContentLengthAsync would be wasted if the called asynchronous process getStringTask has already completed and GetUrlContentLengthAsync doesn't have to wait for the final result.

Inside the calling method the processing pattern continues. The caller might do other work that doesn't depend on the result from GetUrlContentLengthAsync before awaiting that result, or the caller might await immediately. The calling method is waiting for GetUrlContentLengthAsync, and GetUrlContentLengthAsync is waiting for GetStringAsync.

1. GetStringAsync completes and produces a string result. The string result isn't returned by the call to GetStringAsync in the way that you might expect. (Remember that the method already returned a task in step 3.) Instead, the string result is stored in the task that represents the completion of the method, getStringTask. The await operator retrieves the result from getStringTask. The assignment statement assigns the retrieved result to contents.
2. When GetUrlContentLengthAsync has the string result, the method can calculate the length of the string. Then the work of GetUrlContentLengthAsync is also complete, and the waiting event handler can resume. In the full example at the end of the topic, you can confirm that the event handler retrieves and prints the value of the length result. If you are new to asynchronous programming, take a minute to consider the difference between synchronous and asynchronous behavior. A synchronous method returns when its work is complete (step 5), but an async method returns a task value when its work is suspended (steps 3 and 6). When the async method eventually completes its work, the task is marked as completed and the result, if any, is stored in the task.

### Another Simpler Example

**Example 1**

In this example, we are going to take two methods, which are not dependent on each other.

**Code sample**

using System;

namespace AsyncDemoTwo

{

public class Program

{

static void Main(string[] args)

{

Console.WriteLine("Starting Async Demo Two...");

Method1();

Method2();

Console.WriteLine("Finishing Async Demo Two...");

Console.ReadKey();

}

public static async Task Method1()

{

await Task.Run(() =>

{

for (int i = 0; i < 100; i++)

{

Console.WriteLine(" Method 1");

// Do something

Task.Delay(100).Wait();

}

});

}

public static void Method2()

{

for (int i = 0; i < 25; i++)

{

Console.WriteLine(" Method 2");

// Do something

Task.Delay(100).Wait();

}

}

}

}

In the code given above, Method1 and Method2 are not dependent on each other and we are calling from the Main method.

Here, we can clearly see Method1, and Method2 are not waiting for each other.

Now, coming to the second example, suppose we have Method3, which is dependent on Method1

**Example 2**

In this example, Method1 is returning the total length as an integer value and we are passing a parameter as a length in a Method3, which is coming from Method1.

Here, we have to use await keyword before passing a parameter in Method3 and for it, we have to use the async keyword from the calling method.

**Note**: If we are using C# 7 or less, then we cannot use async keyword in the Main method for the console Application because it will give the error below.



We are going to create a new method as callMethod and in this method, we are going to call our all Methods as Method1, Method2, and Method3, respectively (*Refer to C#9 code for demo*).

using System;

namespace AsyncDemoThree

{

public class Program

{

//static void Main(string[] args) // For C#7.

static async Task Main(string[] args) // For C#9.

{

Console.WriteLine("Starting Async Demo Three...");

//callMethod(); // For C#7

await callMethod2(); // For C#9

Console.WriteLine("Finishing Async Demo Three...");

Console.ReadKey();

}

// For C#7.

public static async void callMethod()

{

Task<int> task = Method1();

Method2();

int count = await task;

Method3(count);

}

// For C#9.

public static async Task callMethod2()

{

Task<int> task = Method1();

Method2();

int count = await task;

Method3(count);

}

public static async Task<int> Method1()

{

int count = 0;

await Task.Run(() =>

{

for (int i = 0; i < 100; i++)

{

Console.WriteLine($" Method 1 - {i}");

count += 1;

}

});

return count;

}

public static void Method2()

{

for (int i = 0; i < 25; i++)

{

Console.WriteLine($" Method 2 - {i}");

}

}

public static void Method3(int count)

{

Console.WriteLine("Total count is " + count);

}

}

}

In the code given above, Method3 requires one parameter, which is the return type of Method1. Here, await keyword is playing a vital role for waiting of Method1 task completion.

### Async File I/O

In this example, we are going to read all the characters from a large text file asynchronously and get the total length of all the characters.

using System;

namespace AsyncFileIO

{

public class Program

{

static void Main(string[] args)

{

Console.WriteLine("Starting Async File I/O Demo...");

Task task = new Task(CallMethod);

task.Start();

task.Wait();

Console.WriteLine("Finishing Async File I/O Demo...");

Console.ReadLine();

}

static async void CallMethod()

{

string filePath = @"C:\Temp\TheNotebooksofLeonardoDaVinci.txt";

Task<int> task = ReadFile(filePath);

Console.WriteLine(" Other Work 1");

Console.WriteLine(" Other Work 2");

Console.WriteLine(" Other Work 3");

int length = await task;

Console.WriteLine(" Total length: " + length);

Console.WriteLine(" After work 1");

Console.WriteLine(" After work 2");

}

static async Task<int> ReadFile(string file)

{

int length = 0;

Console.WriteLine(" File reading is stating");

using (StreamReader reader = new StreamReader(file))

{

// Reads all characters from the current position to the end of

// the stream asynchronously and returns them as one string.

string s = await reader.ReadToEndAsync();

//// Sleep just to show the delay.

//Thread.Sleep(1000);

length = s.Length;

}

Console.WriteLine(" File reading is completed");

return length;

}

}

}

In the code given above, we are calling a ReadFile method to read the contents of a text file and get the length of the total characters present in the text file.

In our sampleText.txt, the file contains too many characters, so It will take a long time to read all the characters.

Here, we are using async programming to read all the contents from the file, so it will not wait to get a return value from this method and execute the other lines of code. Still it has to wait for the line of code given below because we are using await keywords, and we are going to use the return value for the line of code given below.

int length = await task;

Console.WriteLine(" Total length: " + length);

Subsequently, other lines of code will be executed sequentially.

Console.WriteLine(" After work 1");

Console.WriteLine(" After work 2");

# Reflection in .NET

## What is .NET Reflection?

.NET Framework's Reflection API allows you to fetch Type (Assembly) information at runtime or programmatically. We can also implement late binding using .NET Reflection. At runtime, Reflection uses the PE file to read the metadata about an assembly. Reflection enables you to use code that was not available at compile time. .NET Reflection allows application to collect information about itself and also manipulate on itself. It can be used effectively to find all the types in an assembly and/or dynamically invoke methods in an assembly. This includes information about the type, properties, methods, and events of an object. With reflection, we can dynamically create an instance of a type, bind the type to an existing object, or get the type from an existing object and invoke its methods or access its fields and properties. We can also access attributes using Reflection. In short, Reflection can be very useful if you don't know much about an assembly.

Using reflection, you can get the kind of information that you will see in the Class Viewer, Object Explorer, or a Class Explorer. You can see all the types in an assembly, their members, their types, and metadata. Here is an example of the Class View in Visual Studio.

Graphical user interface, text, application

Description automatically generated

## Roadmap

The System.Reflection namespace and System.Type class play an important role in .NET Reflection. These two works together and allow you to reflect over many other aspects of a type.

Table

Description automatically generated

## System.Reflection Namespace

System.Reflection Namespace contains classes and interfaces that provide a managed view of loaded types, methods, and fields, with the ability to dynamically create and invoke types; this process is known as Reflection in .NET framework. The following table describes some of the commonly used classes:

|  |  |
| --- | --- |
| Class | Description |
| Assembly | Represents an assembly, which is a reusable, versionable, and self-describing building block of a common language runtime application. This class contains a number of methods that allow you to load, investigate, and manipulate an assembly. |
| Module | Performs reflection on a module. This class allows you to access a given module within a multifile assembly. |
| AssemblyName | This class allows you to discover numerous details behind an assembly's identity. An assembly's identity consists of the following: • Simple name.  • Version number.  • Cryptographic key pair.  • Supported culture |
| EventInfo | This class holds information for a given event. Use the EventInfo class to inspect events and to bind to event handlers FieldInfo. This class holds information for a given field.  Fields are variables defined in the class. FieldInfo provides access to the metadata for a field within a class and provides dynamic set and get functionality for the field. The class is not loaded into memory until invoke or get is called on the object. |
| MemberInfo | The MemberInfo class is the abstract base class for classes used to obtain information about all members of a class (constructors, events, fields, methods, and properties). |
| MethodInfo | This class contains information for a given method. |
| ParameterInfo | This class holds information for a given parameter. |
| PropertyInfo | This class holds information for a given property. |

Before we start using Reflection, it is also necessary to understand the System.Type class.  
  
In order to continue with all the examples given in this article, I am using Car class as an example, it will look like this:

**ICar.cs – Interface**

namespace ReflectionDemos

{

public interface ICar

{

bool IsMoving();

}

}

**Car.cs – Class**

namespace ReflectionDemos

{

public class Car : ICar

{

//public variables

public string Color;

//private variables

//String licensePlate; // e.g. "California 111 222"

//double maxSpeed; // in kilometers per hour

//int startMiles; // Stating odometer reading

//int endMiles; // Ending odometer reading

//double gallons; // Gallons of gas used between the readings

//private vaiables

private int \_speed;

//Speed - read-only property to return the speed

public int Speed

{

get { return \_speed; }

}

//Accelerate - add mph to the speed

public void Accelerate(int accelerateBy)

{

//Adjust the speed

\_speed += accelerateBy;

}

//IsMoving - is the car moving?

public bool IsMoving()

{

//Is the car's speed zero?

if (Speed == 0)

{

return false;

}

else

{

return true;

}

}

//Constructor

public Car()

{

//Set the default values

Color = "White";

\_speed = 0;

}

//Over loaded constructor

public Car(string color, int speed)

{

Color = color;

\_speed = speed;

}

//methods

public double calculateMPG(int startMiles, int endMiles, double gallons)

{

return (endMiles - startMiles) / gallons;

}

}

}

**SportsCar.cs – Class**

namespace ReflectionDemos

{

internal class SportsCar : Car

{

//Constructor

public SportsCar()

{

//Change the default values

Color = "Green";

}

}

}

## The System.Type Class

The System.Type class is the main class for the .NET Reflection functionality to access metadata. The System.Type class is an abstract class and represents a type in the Common Type System (CLS). It represents type declarations: class types, interface types, array types, value types, enumeration types, type parameters, generic type definitions, and open or closed constructed generic types.

 The Type class and its members are used to get information about a type declaration and its members such as constructors, methods, fields, properties, and events of a class, as well as the module and the assembly in which the class is deployed.

There are three ways to obtain a Type reference.

Diagram

Description automatically generated

### Using System.Object.GetType()

This method returns a Type object that represents the type of an object. Obviously, this approach will only work if you have the compile-time knowledge of the type.  
  
**Program.cs –** ObjectGetTypeDemo()

using System;

namespace ReflectionDemos

{

class Program

{

static void Main(string[] args)

{

ObjectGetTypeDemo();

Console.WriteLine();

Console.WriteLine("Press <ENTER> to continue...");

Console.ReadLine();

}

static void ObjectGetTypeDemo()

{

Car c = new Car();

Type t = c.GetType();

Console.WriteLine(t.FullName);

}

}

}

**Output:**

ReflectionsDemos.Car

### Using System.Type.GetType()

Another way of getting Type information, which is more flexible is using the GetType() static method of Type class. This method gets the type with the specified name, performing a case-sensitive search.  
  
The Type.GetType() is an overloaded method and accepts the following parameters:

1. fully qualified string name of the type you are interested in examining
2. exception should be thrown if the type cannot be found
3. establishes the case sensitivity of the string

**Program.cs –** TypeGetTypeDemo()

using System;

namespace ReflectionDemos

{

class Program

{

static void Main(string[] args)

{

TypeGetTypeDemo();

Console.WriteLine();

Console.WriteLine("Press <ENTER> to continue...");

Console.ReadLine();

}

static void TypeGetTypeDemo()

{

Console.WriteLine("Entering TypeGetTypeDemo()...");

// Obtain type information using the static Type.GetType() method.

// (don't throw an exception if Car cannot be found and ignore case).

Type t = Type.GetType("ReflectionDemos.Car", false, true);

Console.WriteLine(t.FullName);

Console.WriteLine("Entering TypeGetTypeDemo()...");

}

}

}

**Output:**

ReflectionsDemos.Car

### Using typeof() C# operator

The final way to obtain a type information is using the C# typeof operator. This operator takes the name of the type as a parameter.

**Program.cs –** TypeofDemo()

namespace ReflectionDemos

{

class Program

{

static void Main(string[] args)

{

TypeofDemo();

Console.WriteLine();

Console.WriteLine("Press <ENTER> to continue...");

Console.ReadLine();

}

static void TypeofDemo()

{

Console.WriteLine("Entering TypeofDemo()...");

// Get the Type using typeof.

Type t = typeof(Car);

Console.WriteLine(t.FullName);

Console.WriteLine("Exiting TypeofDemo()...");

}

}

}

**Output:**

ReflectionsDemos.Car

## Type Properties

The System.Type class defines a number of members that can be used to examine a type's metadata. You can split these properties into three categories,

1. A number of properties retrieve the strings containing various names associated with the class, as shown in the following table:

|  |  |
| --- | --- |
| Property | Returns |
| Name | The name of the data type |
| FullName | The fully qualified name of the data type (including the namespace name) |
| Namespace | The name of the namespace in which the data type is defined |

1. It is also possible to retrieve references to further type objects that represent related classes, as shown in the following table:

|  |  |
| --- | --- |
| Property | Returns Type Reference Corresponding To |
| BaseType | Immediate base type of this type |
| UnderlyingSystemType | The type that this type maps to in the .NET runtime (recall that certain .NET base types actually map to specific predefined types recognized by IL) |

1. A number of Boolean properties indicate whether this type is, for example, a class, an enum, and so on.

|  |  |
| --- | --- |
| Type | Meaning in Life |
| IsAbstract IsArray IsClass IsCOMObject IsEnum IsGenericTypeDefinition IsGenericParameter IsInterface IsPrimitive IsPublic IsNestedPrivate IsNestedPublic IsSealed IsValueType IsPointer | These properties (among others) allow you to discover a number of basic traits about the Type you are referring to (e.g., if it is an abstract method, an array, a nested class, and so forth) |

Here is the example of displaying type information using System.Type class properties:

**Program.cs –** GetTypeProperties()

using System;

using System.Reflection;

using System.Text;

namespace ReflectionDemos

{

class Program

{

static void Main(string[] args)

{

// Modify this line to retrieve details of any other data type

// Get name of type

Type t = typeof(Car);

GetTypeProperties(t);

Console.WriteLine();

Console.WriteLine("Press <ENTER> to continue...");

Console.ReadLine();

}

public static void GetTypeProperties(Type t)

{

Console.WriteLine("Entering GetTypeProperties()...");

StringBuilder OutputText = new StringBuilder();

//properties retrieve the strings

OutputText.AppendLine("Analysis of type " + t.Name);

OutputText.AppendLine("Type Name: " + t.Name);

OutputText.AppendLine("Full Name: " + t.FullName);

OutputText.AppendLine("Namespace: " + t.Namespace);

//properties retrieve references

Type tBase = t.BaseType;

if (tBase != null)

{

OutputText.AppendLine("Base Type: " + tBase.Name);

}

Type tUnderlyingSystem = t.UnderlyingSystemType;

if (tUnderlyingSystem != null)

{

OutputText.AppendLine("UnderlyingSystem Type: " + tUnderlyingSystem.Name);

//OutputText.AppendLine("UnderlyingSystem Type Assembly: " + tUnderlyingSystem.Assembly);

}

//properties retrieve boolean

OutputText.AppendLine("Is Abstract Class: " + t.IsAbstract);

OutputText.AppendLine("Is an Arry: " + t.IsArray);

OutputText.AppendLine("Is a Class: " + t.IsClass);

OutputText.AppendLine("Is a COM Object : " + t.IsCOMObject);

OutputText.AppendLine("\nPUBLIC MEMBERS:");

MemberInfo[] Members = t.GetMembers();

foreach (MemberInfo NextMember in Members)

{

OutputText.AppendLine(NextMember.DeclaringType + " " +

NextMember.MemberType + " " + NextMember.Name);

}

Console.WriteLine(OutputText);

Console.WriteLine("Exiting GetTypeProperties()...");

}

}

}

## Type Methods

Most of the methods of System.Type are used to obtain details of the members of the corresponding data type - constructors, properties, methods, events, and so on. There is a long list of methods exist, but they all follow the same pattern.

|  |  |  |
| --- | --- | --- |
| Returned Type | Methods (The Method with the Plural Name Returns an Array) | Description |
| ConstructorInfo | GetConstructor(), GetConstructors() | These methods allow you to obtain an array representing the items (interface, method, property, etc.) you are interested in. Each method returns a related array (e.g., GetFields() returns a FieldInfo array, GetMethods() returns a MethodInfo array, etc.). Be aware that each of these methods has a singular form (e.g., GetMethod(), GetProperty(), etc.) that allows you to retrieve a specific item by name, rather than an array of all related items. |
| EventInfo | GetEvent(), GetEvents() |
| FieldInfo | GetField(), GetFields() |
| InterfaceInfo | GetInterface(), GetInterfaces() |
| MemberInfo | GetMember(), GetMembers() |
| MethodInfo | GetMethod(), GetMethods() |
| PropertyInfo | GetProperty(), GetProperties() |
|  | FindMembers() | This method returns an array of MemberInfo types based on search criteria. |
| Type | GetType() | This static method returns a Type instance given a string name. |
|  | InvokeMember() | This method allows late binding to a given item. |

For example, two methods retrieve details of the methods of the data type: GetMethod() and GetMethods().

Type t = **typeof**(Car);

MethodInfo[] methods = t.GetMethods();

**foreach** (MethodInfo nextMethod **in** methods)

{

                // etc.

}

### Reflecting on Methods

GetMethod() returns a reference to a System.Reflection.MethodInfo object, which contains details of a method. Searches for the public method with the specified name.

GetMethods() returns an array of such references. The difference is that GetMethods() returns details of all the methods, whereas GetMethod() returns details of just one method with a specified parameter list.  
Both methods have overloads that take an extra parameter, a BindingFlags enumerated value that indicates which members should be returned - for example, whether to return public members, instance members, static members, and so on.

MethodInfo is derived from the abstract class MethodBase, which inherits MemberInfo. Thus, the properties and methods defined by all three of these classes are available for your use.

Diagram

Description automatically generated

**Program.cs –** GetMethod() and GetMethods()

using System;

using System.Reflection;

using System.Text;

namespace ReflectionDemos

{

class Program

{

static void Main(string[] args)

{

// Type Methods.

// Get name of type

Type t2 = typeof(Car);

GetMethod(t2);

GetMethods(t2);

Console.WriteLine("Press <ENTER> to continue...");

Console.ReadLine();

}

public static void GetMethods(Type t)

{

Console.WriteLine("\*\*\*\*\* Methods \*\*\*\*\*");

Console.WriteLine("Entering GetMethods()...");

MethodInfo[] mi = t.GetMethods();

foreach (MethodInfo m in mi)

Console.WriteLine("->{0}", m.Name);

Console.WriteLine("Exiting GetMethods()...");

}

// Display method name of type.

public static void GetMethod(Type t)

{

Console.WriteLine("\*\*\*\*\* Method \*\*\*\*\*");

Console.WriteLine("Entering GetMethod()...");

//This searches for name is case-sensitive.

//The search includes public static and public instance methods.

MethodInfo mi = t.GetMethod("IsMoving");

Console.WriteLine("->{0}", mi.Name);

Console.WriteLine("Exiting GetMethod()...");

}

}

}

## Reflecting on Fields and Properties

Behavior of the Type.GetField() and Type.GetFields() is exactly similar to above two methods except Type.GetField() returns to references of System.Reflection.MethodInfo and Type.GetFields() returns to references of System.Reflection.MethodInfo array. Similarly Type.GetProperty() and Type.GetProperties() too.

The logic to display a type's properties is similar:

**Program.cs –** GetFields() and GetField()

using System;

using System.Reflection;

using System.Text;

namespace ReflectionDemos

{

class Program

{

static void Main(string[] args)

{

// Type Methods.

// Get name of type

Type t3 = typeof(Car);

GetFields(t3);

GetProperties(t3);

Console.WriteLine();

Console.WriteLine("Press <ENTER> to continue...");

Console.ReadLine();

}

// Display field names of type.

public static void GetFields(Type t)

{

Console.WriteLine("\*\*\*\*\* Fields \*\*\*\*\*");

Console.WriteLine("Entering GetFields()...");

FieldInfo[] fi = t.GetFields();

foreach (FieldInfo field in fi)

Console.WriteLine("->{0}", field.Name);

Console.WriteLine("Exiting GetFields()...");

}

// Display property names of type.

public static void GetProperties(Type t)

{

Console.WriteLine("\*\*\*\*\* Properties \*\*\*\*\*");

Console.WriteLine("Entering GetProperties()...");

PropertyInfo[] pi = t.GetProperties();

foreach (PropertyInfo prop in pi)

Console.WriteLine("->{0}", prop.Name);

Console.WriteLine("Exiting GetProperties()...");

}

}

}

## Reflecting on Implemented Interfaces

GetInterfaces() returns an array of System.Types. his should make sense given that interfaces are, indeed, types:

**Program.cs –** GetInterfaces()

using System;

using System.Reflection;

using System.Text;

namespace ReflectionDemos

{

class Program

{

static void Main(string[] args)

{

// Get name of type

Type t = typeof(Car);

// Reflecting on Interfaces.

GetInterfaces(t);

Console.WriteLine();

Console.WriteLine("Press <ENTER> to continue...");

Console.ReadLine();

}

// Display implemented interfaces.

public static void GetInterfaces(Type t)

{

Console.WriteLine("\*\*\*\*\* Interfaces \*\*\*\*\*");

Console.WriteLine("Entering GetInterfaces()...");

Type[] ifaces = t.GetInterfaces();

foreach (Type i in ifaces)

Console.WriteLine("->{0}", i.Name);

Console.WriteLine("Exiting GetInterfaces()...");

}

}

}

## Reflecting on Method Parameters and Return Values

To play with method parameters and their return types, we first need to build MethodInfo[] array using GetMethods() function. The MethodInfo type provides the ReturnType property and GetParameters() method for these very tasks.

**Program.cs –** GetParametersInfo()

using System;

using System.Reflection;

using System.Text;

namespace ReflectionDemos

{

class Program

{

static void Main(string[] args)

{

// Get name of type

Type t = typeof(Car);

// Reflecting on Methods Parameters and Return Values.

GetParametersInfo(t);

Console.WriteLine();

Console.WriteLine("Press <ENTER> to continue...");

Console.ReadLine();

}

//Display Method return Type and paralmeters list

public static void GetParametersInfo(Type t)

{

Console.WriteLine("\*\*\*\*\* GetParametersInfo \*\*\*\*\*");

Console.WriteLine("Entering GetParametersInfo()...");

MethodInfo[] mi = t.GetMethods();

foreach (MethodInfo m in mi)

{

// Get return value.

string retVal = m.ReturnType.FullName;

StringBuilder paramInfo = new StringBuilder();

paramInfo.Append("(");

// Get params.

foreach (ParameterInfo pi in m.GetParameters())

{

paramInfo.Append(string.Format("{0} {1} ", pi.ParameterType, pi.Name));

}

paramInfo.Append(")");

// Now display the basic method sig.

Console.WriteLine("->{0} {1} {2}", retVal, m.Name, paramInfo);

}

Console.WriteLine("Exiting GetParametersInfo()...");

}

}

}

## Reflecting on Constructor

GetConstractors() function returns an array of ConstractorInfo elements, which we can use to get constructors' information.

**Program.cs –** GetConstructorsInfo()

using System;

using System.Reflection;

using System.Text;

namespace ReflectionDemos

{

class Program

{

static void Main(string[] args)

{

// Get name of type

Type t = typeof(Car);

// Reflecting on Constructor.

GetConstructorsInfo(t);

Console.WriteLine();

Console.WriteLine("Press <ENTER> to continue...");

Console.ReadLine();

}

// Display method names of type.

public static void GetConstructorsInfo(Type t)

{

Console.WriteLine("\*\*\*\*\* ConstructorsInfo \*\*\*\*\*");

Console.WriteLine("Entering ConstructorsInfo()...");

ConstructorInfo[] ci = t.GetConstructors();

foreach (ConstructorInfo c in ci)

Console.WriteLine(c.ToString());

Console.WriteLine("Exiting ConstructorsInfo()...");

}

}

}

## Assembly Class

System.Reflection namespace provide a class called Assembly. We can use this Assembly class to fetch the information about the assembly and manipulate it; this class allows us to load modules and assemblies at run time. Assembly class contacts with PE file to fetch the metadata information about the assembly at runtime. Once we load an assembly using this Assembly class, we can search the type information within the assembly. It is also possible to create instance of types return by the Assembly class.

### Dynamically loading an Assembly

Assembly Class provides following methods to load an assembly at runtime,

1. *Load ()*This static overloaded method takes the assembly name as input parameter and searched the given assembly name in the system.
2. LoadFrom ()This static overloaded method take complete path of the an assembly, it will directly look into that particular location instead of searching in the system.
3. *GetExecutingAssembly ()*  
   Assembly class also provide another method to obtain the currently running assembly information using GetExecutingAssembly() methods. This method is not overloaded one.
4. *GetTypes()*Assembly class also provide a nice feature called GetTypes Method which allows you to obtain details of all the types that are defined in the corresponding assembly.
5. *GetCustomAttributes()*This static overloaded method gets the attributes attached to the assembly. You can also call GetCustomAttributes() specifying a second parameter, which is a Type object that indicates the attribute class in which you are interested.

**Program.cs –** LoadAsm()

using System;

using System.Reflection;

using System.Text;

namespace ReflectionDemos

{

class Program

{

static void Main(string[] args)

{

LoadAsm();

Console.WriteLine();

Console.WriteLine("Press <ENTER> to continue...");

Console.ReadLine();

}

static void LoadAsm()

{

Console.WriteLine("Entering LoadAsm()...");

Assembly objAssembly;

// You must supply a valid fully qualified assembly name here.

//objAssembly = Assembly.Load("mscorlib,4.0.0.0,Neutral");

// Loads an assembly using its file name

//objAssembly = Assembly.LoadFrom(@"C:\Windows\Microsoft.NET\Framework\v4.0.30319\CasPol.exe");

objAssembly = Assembly.LoadFrom(@"C:\Program Files\dotnet\packs\Microsoft.NETCore.App.Ref\6.0.1\ref\net6.0\mscorlib.dll");

//this loads currnly running process assembly

//objAssembly = Assembly.GetExecutingAssembly();

Type[] Types = objAssembly.GetTypes();

// Display all the types contained in the specified assembly.

foreach (Type objType in Types)

{

Console.WriteLine(objType.Name.ToString());

}

//fetching custom attributes within an assembly

Attribute[] arrayAttributes =

Attribute.GetCustomAttributes(objAssembly);

// assembly1 is an Assembly object

foreach (Attribute attrib in arrayAttributes)

{

Console.WriteLine(attrib.TypeId);

}

Console.WriteLine("Exiting LoadAsm()...");

} }

}

## Late Binding

Late binding is the powerful tool in .NET Reflection, which allows you to create an instance of a given type and invoke its members at runtime without having compile-time knowledge of its existence; this technique is also called dynamic invocation. This technique is useful when working with objects that are does not provide details at compile time. In this technique, developers are responsible for passing the correct signature of methods before invoking them, otherwise it will throw an error. It is very important to take the right decision when using this approach. Use of late binding may also impact the performance of your application.

**Program.cs –** LateBinding()

using System;

using System.Reflection;

using System.Text;

namespace ReflectionDemos

{

class Program

{

static void Main(string[] args)

{

LateBinding();

Console.WriteLine();

Console.WriteLine("Press <ENTER> to continue...");

Console.ReadLine();

}

// Late binding.

static void LateBinding()

{

Console.WriteLine("Entering LateBinding()...");

Assembly objAssembly;

// Loads an assembly

objAssembly = Assembly.GetExecutingAssembly();

//get the class type information in which late binding applied

Type classType = objAssembly.GetType("ReflectionDemos.Car");

//create the instance of class using System.Activator class

object obj = Activator.CreateInstance(classType);

//get the method information

MethodInfo mi = classType.GetMethod("IsMoving");

//Late Binding using Invoke method without parameters

bool isCarMoving;

isCarMoving = (bool)mi.Invoke(obj, null);

if (isCarMoving)

{

Console.WriteLine("Car Moving Status is : Moving");

}

else

{

Console.WriteLine("Car Moving Status is : Not Moving");

}

//Late Binding with parameters

object[] parameters = new object[3];

parameters[0] = 32456;//parameter 1 startMiles

parameters[1] = 32810;//parameter 2 end Miles

parameters[2] = 10.6;//parameter 3 gallons

mi = classType.GetMethod("calculateMPG");

double MilesPerGallon;

MilesPerGallon = (double)mi.Invoke(obj, parameters);

Console.WriteLine("Miles per gallon is : " + MilesPerGallon);

Console.WriteLine("Exiting LateBinding()...");

}

}

}

# Design Pattern – Factory Method

Design patterns are reusable solutions to common problems in software design and factory design pattern is one of the common design patterns. In this article, we will learn what a factory design pattern is and how to implement a factory pattern in C#. We will also learn when to apply a factory design pattern in C# and .NET real-world applications.

## Factory design pattern

Before learning Factory Method Pattern, I just want to share a little about "Gang of Four (GoF)" to which the Factory Method pattern belongs.

## **Who is the Gang of Four?**

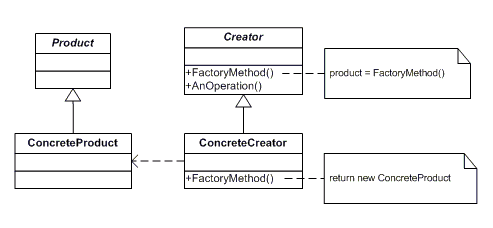
The Gang of Four is the authors of the book, "Design Patterns: Elements of Reusable Object-Oriented Software". This important book describes various development techniques and pitfalls in addition to providing 23 object-oriented programming design patterns. The four authors are Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides.

Now, let's move on to the Factory Method design pattern:

Factory Method is a Design Pattern which defines an interface for creating an object but lets the classes that implement the interface decide which class to instantiate. Factory Pattern lets a class postpone instantiation to sub-classes. The factory pattern is used to replace class constructors, abstracting the process of object generation so that the type of the object instantiated can be determined at run-time.

## Where to Use It?

It is tedious when the client needs to specify the class name while creating the objects. So, to resolve this problem, we can use the Factory Method design pattern. It provides the client with a simple way to create the object.



The classes and objects participating in the above UML class diagram are as follows:

1. Product

This defines the interface of objects the factory method creates

1. ConcreteProduct

This is a class that implements the Product interface.

1. Creator

This is an abstract class and declares the factory method, which returns an object of type Product.

This may also define a default implementation of the factory method that returns a default ConcreteProduct object.

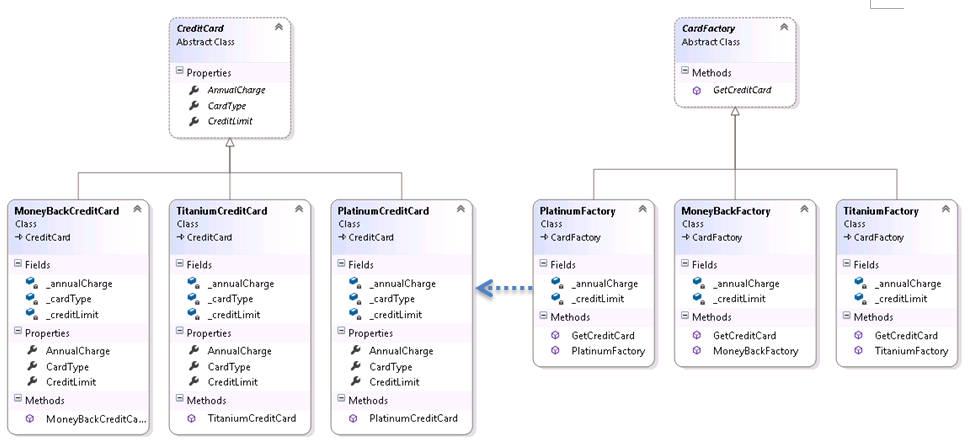
This may call the factory method to create a Product object.

1. ConcreteCreator

This is a class that implements the Creator class and overrides the factory method to return an instance of a ConcreteProduct.

## Factory design pattern in real-world example

Assume you have three different cards which are considered here as classes MoneyBack, Titanium and Platinum, all of them implement abstract class CreditCard. You need to instantiate one of these classes, but you don't know which of them, it depends on the user. This is a perfect scenario for the Factory Method design pattern.



### Who is what?

The classes and objects participating in the above class diagram can be identified as follows:

1. **Product** – CreditCard
2. **ConcreteProduct** – MoneyBackCreditCard, TitaniumCreditCard, PlatinumCreditCard
3. **Creator** – CardFactory
4. **ConcreteCreator** – MoneyBackCardFactory, TitaniumCardFactory, PlatinumCardFactory

Here are the code blocks for each participant:

**Product –** CreditCard

namespace FactoryMethodDP

{

/// <summary>

/// The 'Product' Abstract Class

/// </summary>

abstract class CreditCard

{

public abstract string CardType { get; }

public abstract int CreditLimit { get; set; }

public abstract int AnnualCharge { get; set; }

}

}

**ConcreteProduct –** MoneyBackCreditCard

namespace FactoryMethodDP

{

/// <summary>

/// A 'ConcreteProduct' class

/// </summary>

class MoneyBackCreditCard : CreditCard

{

private readonly string \_cardType;

private int \_creditLimit;

private int \_annualCharge;

public MoneyBackCreditCard(int creditLimit, int annualCharge)

{

\_cardType = "MoneyBack";

\_creditLimit = creditLimit;

\_annualCharge = annualCharge;

}

public override string CardType

{

get { return \_cardType; }

}

public override int CreditLimit

{

get { return \_creditLimit; }

set { \_creditLimit = value; }

}

public override int AnnualCharge

{

get { return \_annualCharge; }

set { \_annualCharge = value; }

}

}

}

**ConcreteProduct –** TitaniumCreditCard

namespace FactoryMethodDP

{

/// <summary>

/// A 'ConcreteProduct' class

/// </summary>

class TitaniumCreditCard : CreditCard

{

private readonly string \_cardType;

private int \_creditLimit;

private int \_annualCharge;

public TitaniumCreditCard(int creditLimit, int annualCharge)

{

\_cardType = "Titanium";

\_creditLimit = creditLimit;

\_annualCharge = annualCharge;

}

public override string CardType

{

get { return \_cardType; }

}

public override int CreditLimit

{

get { return \_creditLimit; }

set { \_creditLimit = value; }

}

public override int AnnualCharge

{

get { return \_annualCharge; }

set { \_annualCharge = value; }

}

}

}

**ConcreteProduct –** PlatinumCreditCard

namespace FactoryMethodDP

{

/// <summary>

/// A 'ConcreteProduct' class

/// </summary>

class PlatinumCreditCard : CreditCard

{

private readonly string \_cardType;

private int \_creditLimit;

private int \_annualCharge;

public PlatinumCreditCard(int creditLimit, int annualCharge)

{

\_cardType = "Platinum";

\_creditLimit = creditLimit;

\_annualCharge = annualCharge;

}

public override string CardType

{

get { return \_cardType; }

}

public override int CreditLimit

{

get { return \_creditLimit; }

set { \_creditLimit = value; }

}

public override int AnnualCharge

{

get { return \_annualCharge; }

set { \_annualCharge = value; }

}

}

}

**Creator –** CardFactory

namespace FactoryMethodDP

{

/// <summary>

/// The 'Creator' Abstract Class

/// </summary>

// This could be an interface.

abstract class CardFactory

{

public abstract CreditCard GetCreditCard();

}

}

**ConcreteCreator –** MoneyBackFactory

namespace FactoryMethodDP

{

/// <summary>

/// A 'ConcreteCreator' class

/// </summary>

class MoneyBackFactory : CardFactory

{

private int \_creditLimit;

private int \_annualCharge;

public MoneyBackFactory(int creditLimit, int annualCharge)

{

\_creditLimit = creditLimit;

\_annualCharge = annualCharge;

}

public override CreditCard GetCreditCard()

{

return new MoneyBackCreditCard(\_creditLimit, \_annualCharge);

}

}

}

**ConcreteCreator –** TitaniumFactory

namespace FactoryMethodDP

{

/// <summary>

/// A 'ConcreteCreator' class

/// </summary>

class TitaniumFactory : CardFactory

{

private int \_creditLimit;

private int \_annualCharge;

public TitaniumFactory(int creditLimit, int annualCharge)

{

\_creditLimit = creditLimit;

\_annualCharge = annualCharge;

}

public override CreditCard GetCreditCard()

{

return new TitaniumCreditCard(\_creditLimit, \_annualCharge);

}

}

}

**ConcreteCreator –** PlatinumFactory

namespace FactoryMethodDP

{

/// <summary>

/// A 'ConcreteCreator' class

/// </summary>

class PlatinumFactory : CardFactory

{

private int \_creditLimit;

private int \_annualCharge;

public PlatinumFactory(int creditLimit, int annualCharge)

{

\_creditLimit = creditLimit;

\_annualCharge = annualCharge;

}

public override CreditCard GetCreditCard()

{

return new PlatinumCreditCard(\_creditLimit, \_annualCharge);

}

}

}

**Factory Pattern Client Program**

using System;

namespace FactoryMethodDP

{

public class Program

{

public static void Main(string[] args)

{

Console.WriteLine("Starting Factory Method Design Pattern Demo...");

RunDemo();

Console.WriteLine("Press any key to continue...");

Console.ReadKey();

}

static void RunDemo()

{

bool toEnd = false;

do

{

CardFactory factory = null;

Console.Write("Enter the card type you would like to create (Moneyback, Titanium or Platinum): ");

string card = Console.ReadLine();

if (string.IsNullOrEmpty(card))

continue;

switch (card.ToLower())

{

case "quit":

case "exit":

toEnd = true;

break;

case "moneyback":

factory = new MoneyBackFactory(50000, 0);

break;

case "titanium":

factory = new TitaniumFactory(100000, 500);

break;

case "platinum":

factory = new PlatinumFactory(500000, 1000);

break;

default:

Console.WriteLine("Invalid card type selected. Please retry...");

continue;

//break;

}

if (!toEnd)

{

CreditCard creditCard = factory.GetCreditCard();

Console.WriteLine("\nYour card details are below : \n");

Console.WriteLine("Card Type: {0}\nCredit Limit: {1}\nAnnual Charge: {2}",

creditCard.CardType, creditCard.CreditLimit, creditCard.AnnualCharge);

}

} while (!toEnd);

}

}

}

# REST – REpresentational State Transfer

REST is an acronym for **RE**presentational **S**tate **T**ransfer and an architectural style for **distributed hypermedia systems**. Roy Fielding first presented it in 2000 in his famous dissertation.

Like other architectural styles, REST has its guiding principles and constraints. These principles must be satisfied if a service interface needs to be referred to as **RESTful**.

A Web API (or Web Service) conforming to the REST architectural style is a REST API.

## Guiding Principles of REST

The six guiding principles or constraints of the RESTful architecture are:

### Uniform Interface

By applying the principle of generality to the components interface, we can simplify the overall system architecture and improve the visibility of interactions.

Multiple architectural constraints help in obtaining a uniform interface and guiding the behavior of components.

The following four constraints can achieve a uniform REST interface:

* **Identification of resources** – The interface must uniquely identify each resource involved in the interaction between the client and the server.
* **Manipulation of resources through representations** – The resources should have uniform representations in the server response. API consumers should use these representations to modify the resources state in the server.
* **Self-descriptive messages** – Each resource representation should carry enough information to describe how to process the message. It should also provide information of the additional actions that the client can perform on the resource.
* **Hypermedia as the engine of application state** – The client should have only the initial URI of the application. The client application should dynamically drive all other resources and interactions with the use of hyperlinks.

### Client-server

The client-server design pattern enforces the **separation of concerns**, which helps the client and the server components evolve independently.

By separating the user interface concerns (client) from the data storage concerns (server), we improve the portability of the user interface across multiple platforms and improve scalability by simplifying the server components.

While the client and the server evolve, we have to make sure that the interface/contract between the client and the server does not break.

### Stateless

Statelessness mandates that each request from the client to the server must contain all of the information necessary to understand and complete the request.

The server cannot take advantage of any previously stored context information on the server.

For this reason, the client application must entirely keep the session state.

### Cacheable

The cacheable constraint requires that a response should implicitly or explicitly label itself as cacheable or non-cacheable.

If the response is cacheable, the client application gets the right to reuse the response data later for equivalent requests and a specified period.

### Layered system

The layered system style allows an architecture to be composed of hierarchical layers by constraining component behavior.

For example, in a layered system, each component cannot see beyond the immediate layer they are interacting with.

### Code on demand (optional)

REST also allows client functionality to extend by downloading and executing code in the form of applets or scripts.

The downloaded code simplifies clients by reducing the number of features required to be pre-implemented. Servers can provide part of features delivered to the client in the form of code, and the client only needs to execute the code.

## What is a Resource?

The key **abstraction of information** in REST is a resource. Any information that we can name can be a resource. For example, a REST resource can be a document or image, a temporal service, a collection of other resources, or a non-virtual object (e.g., a person).

The state of the resource, at any particular time, is known as the **resource representation**.

The resource representations are consisted of:

* the **data**
* the **metadata** describing the data
* and the **hypermedia links** that can help the clients in transition to the next desired state.

A REST API consists of an assembly of interlinked resources. This set of resources is known as the REST API’s ***resource model***.

### Resource Identifiers

REST uses resource identifiers to identify each resource involved in the interactions between the client and the server components.

### Hypermedia

The data format of a representation is known as a media type. The media type identifies a specification that defines how a representation is to be processed.

**A RESTful API looks like hypertext.** Every addressable unit of information carries an address, either explicitly (e.g., link and id attributes) or implicitly (e.g., derived from the media type definition and representation structure).

Hypertext (or hypermedia) means the **simultaneous presentation of information and controls** such that the information becomes the affordance through which the user (or automaton) obtains choices and selects actions.

Remember that hypertext does not need to be HTML (or XML or JSON) on a browser. Machines can follow links when they understand the data format and relationship types.

— Roy Fielding

### Self-descriptive

Further,**resource representations shall be self-descriptive**: the client does not need to know if a resource is an employee or a device. It should act based on the media type associated with the resource.

So, in practice, we will create lots of **custom media types** – usually one media type associated with one resource.

Every media type defines a default processing model. For example, HTML defines a rendering process for hypertext and the browser behavior around each element.

Media Types have no relation to the resource methods GET/PUT/POST/DELETE/… other than the fact that some media type elements will define a process model that goes like “anchor elements with an href attribute create a hypertext link that, when selected, invokes a retrieval request (GET) on the URI corresponding to the CDATA-encoded href attribute.”

## Resource Methods

Another important thing associated with REST is **resource methods**. These resource methods are used to perform the desired transition between two states of any resource.

A large number of people wrongly relate resource methods to HTTP methods (i.e., GET/PUT/POST/DELETE). Roy Fielding has never mentioned any recommendation around which method to be used in which condition. All he emphasizes is that it should be a **uniform interface**.

For example, if we decide that the application APIs will use HTTP POST for updating a resource – rather than most people recommend HTTP PUT – it’s all right. Still, the application interface will be RESTful.

Ideally, everything needed to transition the resource state shall be part of the resource representation – including all the supported methods and what form they will leave the representation.

We should enter a REST API with no prior knowledge beyond the initial URI (a bookmark) and a set of standardized media types appropriate for the intended audience (i.e., expected to be understood by any client that might use the API).

From that point on, all application state transitions must be driven by the client selection of server-provided choices present in the received representations or implied by the user’s manipulation of those representations.

The transitions may be determined (or limited by) the client’s knowledge of media types and resource communication mechanisms, both of which may be improved on the fly (e.g., code-on-demand). [Failure here implies that out-of-band information is driving interaction instead of hypertext.]

## REST and HTTP are not same

Many people prefer to compare HTTP with REST. **REST and HTTP are not the same.**

**REST != HTTP**

Though REST also intends to make the web (internet) more streamlined and standard, Roy fielding advocates using REST principles more strictly. And that’s from where people try to start comparing REST with the web.

Roy fielding, in his dissertation, has nowhere mentioned any implementation direction – including any protocol preference or even HTTP. Till the time, we are honoring the six guiding principles of REST, which we can call our interface – RESTful.

## Summary

In simple words, in the REST architectural style, data and functionality are considered resources and are accessed using **Uniform Resource Identifiers** (URIs).

The resources are acted upon by using a set of simple, well-defined operations. Also, the resources have to be decoupled from their representation so that clients can access the content in various formats, such as HTML, XML, plain text, PDF, JPEG, JSON, and others.

The clients and servers exchange representations of resources by using a standardized interface and protocol. Typically HTTP is the most used protocol, but REST does not mandate it.

Metadata about the resource is made available and used to control caching, detect transmission errors, negotiate the appropriate representation format, and perform authentication or access control.

And most importantly, every interaction with the server must be stateless.

All these principles help RESTful applications to be simple, lightweight, and fast.

# REST – HATEOAS

**HATEOAS (Hypermedia as the Engine of Application State)** is a constraint of the REST application architecture. HATEOAS keeps the REST style architecture unique from most other network application architectures.

The term “**hypermedia**” refers to any content that contains links to other forms of media such as images, movies, and text.

REST architectural style lets us use the hypermedia links in the API response contents. It allows the client to dynamically navigate to the appropriate resources by traversing the hypermedia links.

Navigating hypermedia links is conceptually the same as browsing through web pages by clicking the relevant hyperlinks to achieve a final goal.

For example, the given below JSON response may be from an API like:

HTTP GET http://api.domain.com/management/departments/10

{

"departmentId": 10,

"departmentName": "Administration",

"locationId": 1700,

"managerId": 200,

"links": [

{

"href": "10/employees",

"rel": "employees",

"type" : "GET"

}

]

}

In the preceding example, the response returned by the server contains hypermedia links to employee resources 10/employees which can be traversed by the client to read employees belonging to the department.

The advantage of the above approach is that hypermedia links returned from the server drive the application’s state and not the other way around.

JSON does not have any universally accepted format for representing links between two resources. We may choose to send in the response body or decide to send links in HTTP response headers.

HTTP/1.1 200 OK

...

Link: <10/employees>; rel="employees"

Both are good solutions.

## How to Implement HATEOAS

In the real world, when we visit a website – we hit its homepage. The homepage presents some snapshots and links to other sections of websites. We click on the links and get more information and related links relevant to the context.

Like a human’s interaction with a website, a **REST client hits an initial API URI and uses the server-provided links to access the resources it needs and discover available actions dynamically**.

The client need not have prior knowledge of the service or the different steps involved in a workflow. Additionally, the **clients no longer have to hardcode the URI structures for various resources**. HATEOAS allows the server to make URI changes as the API evolves without breaking the clients.

Above API interaction is possible using HATEOAS only.

# ASP.NET Core

ASP.NET Core is a cross-platform, high-performance, open-source framework for building modern, cloud-enabled, Internet-connected apps. With ASP.NET Core, you can:

* Build web apps and services, Internet of Things (IoT) apps, and mobile backends.
* Use your favorite development tools on Windows, macOS, and Linux.
* Deploy to the cloud or on-premises.
* Run on .NET Core.

## Why choose ASP.NET Core?

Millions of developers use or have used ASP.NET 4.x to create web apps. ASP.NET Core is a redesign of ASP.NET 4.x, including architectural changes that result in a leaner, more modular framework.

ASP.NET Core provides the following benefits:

* A unified story for building web UI and web APIs.
* Architected for testability.
* Razor Pages makes coding page-focused scenarios easier and more productive.
* Blazor lets you use C# in the browser alongside JavaScript. Share server-side and client-side app logic all written with .NET.
* Ability to develop and run on Windows, macOS, and Linux.
* Open-source and community-focused.
* Integration of modern, client-side frameworks and development workflows.
* Support for hosting Remote Procedure Call (RPC) services using gRPC.
* A cloud-ready, environment-based configuration system.
* Built-in dependency injection.
* A lightweight, high-performance, and modular HTTP request pipeline.
* Ability to host on the following:
  + Kestrel
  + IIS
  + HTTP.sys
  + Nginx
  + Apache
  + Docker
* Side-by-side versioning.
* Tooling that simplifies modern web development.

## Build web APIs and web UI using ASP.NET Core MVC

ASP.NET Core MVC provides features to build web APIs and web apps:

* The Model-View-Controller (MVC) pattern helps make your web APIs and web apps testable.
* Razor Pages is a page-based programming model that makes building web UI easier and more productive.
* Razor markup provides a productive syntax for Razor Pages and MVC views.
* Tag Helpers enable server-side code to participate in creating and rendering HTML elements in Razor files.
* Built-in support for multiple data formats and content negotiation lets your web APIs reach a broad range of clients, including browsers and mobile devices.
* Model binding automatically maps data from HTTP requests to action method parameters.
* Model validation automatically performs client-side and server-side validation.

## Client-side development

ASP.NET Core integrates seamlessly with popular client-side frameworks and libraries, including Blazor, Angular, React, and Bootstrap.

## Framework selection

The following table compares ASP.NET Core to ASP.NET 4.x.

| **FRAMEWORK SELECTION** | | |
| --- | --- | --- |
| **ASP.NET Core** | **ASP.NET 4.x** |
| Build for Windows, macOS, or Linux | Build for Windows |
| Razor Pages is the recommended approach to create a Web UI as of ASP.NET Core 2.x. See also MVC, Web API, and SignalR. | Use Web Forms,  SignalR, MVC, Web API, WebHooks, or Web Pages |
| Multiple versions per machine | One version per machine |
| Develop with Visual Studio, Visual Studio for Mac, or Visual Studio Code using C# or F# | Develop with Visual Studio using C#, VB, or F# |
| Higher performance than ASP.NET 4.x | Good performance |
| Use .NET Core runtime | Use .NET Framework runtime |

## Demo: Create an ASP.NET Core Web App – Visual Studio

New -> Project -> ASP.NET Core Web App

## Demo: Create an ASP.NET Core Web App – Visual Studio Code

### Create a web app project

Open a command shell, and enter the following command:

dotnet new webapp -o aspnetcoreapp

The preceding command:

* Creates a new web app.
* The -o aspnetcoreapp parameter creates a directory named aspnetcoreapp with the source files for the app.

### Trust the development certificate

Trust the HTTPS development certificate:

**Windows**

dotnet dev-certs https --trust



**macOS**

dotnet dev-certs https --trust

The preceding command displays the following message:

Trusting the HTTPS development certificate was requested. If the certificate is not already trusted, we will run the following command: 'sudo security add-trusted-cert -d -r trustRoot -k /Library/Keychains/System.keychain <<certificate>>'

This command might prompt you for your password to install the certificate on the system keychain. Enter your password if you agree to trust the development certificate.

**Linux**

See the documentation for your Linux distribution on how to trust the HTTPS development certificate.

### Run the app

Run the following commands:

cd aspnetcoreapp

dotnet watch run

After the command shell indicates that the app has started, browse to https://localhost:5001.

## ASP.NET Core Fundamentals

### Program.cs

ASP.NET Core apps created with the web templates contain the application startup code in the Program.cs file. The Program.cs file is where:

* Services required by the app are configured.
* The app's request handling pipeline is defined as a series of middleware components.

The app startup code in the above example code supports:

* Razor Pages
* MVC controllers with views

It can also support *Web API with controllers* (to be discussed later).

### Dependency injection (services)

ASP.NET Core includes dependency injection (DI) that makes configured services available throughout an app. Services are added to the DI container with WebApplicationBuilder.Services, builder.Services in the preceding code. When the WebApplicationBuilder is instantiated, many framework-provided services are added. builder is a WebApplicationBuilder in the following code:

var builder = WebApplication.CreateBuilder(args);

// Add services to the container.

builder.Services.AddRazorPages();

builder.Services.AddControllersWithViews();

var app = builder.Build();

In the preceding highlighted code, builder has configuration, logging, and many other services added to the DI container.

The following code adds Razor Pages, MVC controllers with views, and a custom DbContext to the DI container:

using Microsoft.EntityFrameworkCore;

using RazorPagesMovie.Data;

var builder = WebApplication.CreateBuilder(args);

// Add services to the container.

builder.Services.AddRazorPages();

builder.Services.AddControllersWithViews();

builder.Services.AddDbContext<RazorPagesMovieContext>(options =>

options.UseSqlServer(builder.Configuration.GetConnectionString("RPMovieContext")));

var app = builder.Build();

Services are typically resolved from DI using constructor injection. The DI framework provides an instance of this service at runtime.

The following code uses constructor injection to resolve the database context and logger from DI:

public class IndexModel : PageModel

{

private readonly RazorPagesMovieContext \_context;

private readonly ILogger<IndexModel> \_logger;

public IndexModel(RazorPagesMovieContext context, ILogger<IndexModel> logger)

{

\_context = context;

\_logger = logger;

}

public IList<Movie> Movie { get;set; }

public async Task OnGetAsync()

{

\_logger.LogInformation("IndexModel OnGetAsync.");

Movie = await \_context.Movie.ToListAsync();

}

}

### Middleware

The request handling pipeline is composed as a series of middleware components. Each component performs operations on an HttpContext and either invokes the next middleware in the pipeline or terminates the request.

By convention, a middleware component is added to the pipeline by invoking a Use{Feature} extension method. Middleware added to the app is highlighted in the following code:

var builder = WebApplication.CreateBuilder(args);

// Add services to the container.

builder.Services.AddRazorPages();

builder.Services.AddControllersWithViews();

var app = builder.Build();

// Configure the HTTP request pipeline.

if (!app.Environment.IsDevelopment())

{

app.UseExceptionHandler("/Error");

app.UseHsts();

}

app.UseHttpsRedirection();

app.UseStaticFiles();

app.UseAuthorization();

app.MapGet("/hi", () => "Hello!");

app.MapDefaultControllerRoute();

app.MapRazorPages();

app.Run();

### Host

On startup, an ASP.NET Core app builds a host. The host encapsulates all of the app's resources, such as:

* An HTTP server implementation
* Middleware components
* Logging
* Dependency injection (DI) services
* Configuration

There are three different hosts:

* .NET WebApplication Host
* .NET Generic Host
* ASP.NET Core Web Host

The .NET Minimal Host is recommended and used in all the ASP.NET Core templates. The Minimal and Generic hosts share many of the same interfaces and classes. The ASP.NET Core Web Host is available only for backwards compatibility.

The following example instantiates a WebApplication Host:

var builder = WebApplication.CreateBuilder(args);

// Add services to the container.

builder.Services.AddRazorPages();

builder.Services.AddControllersWithViews();

var app = builder.Build();

The WebApplicationBuilder.Build method configures a host with a set of default options, such as:

* Use Kestrel as the web server and enable IIS integration.
* Load configuration from appsettings.json, environment variables, command line arguments, and other configuration sources.
* Send logging output to the console and debug providers.

### Non-web scenarios

The Generic Host allows other types of apps to use cross-cutting framework extensions, such as logging, dependency injection (DI), configuration, and app lifetime management. For more information, see .NET Generic Host in ASP.NET Core and Background tasks with hosted services in ASP.NET Core.

#### Servers

An ASP.NET Core app uses an HTTP server implementation to listen for HTTP requests. The server surfaces requests to the app as a set of request features composed into an HttpContext.

**Windows**

ASP.NET Core provides the following server implementations:

* Kestrel is a cross-platform web server. Kestrel is often run in a reverse proxy configuration using IIS. In ASP.NET Core 2.0 or later, Kestrel can be run as a public-facing edge server exposed directly to the Internet.
* IIS HTTP Server is a server for Windows that uses IIS. With this server, the ASP.NET Core app and IIS run in the same process.
* HTTP.sys is a server for Windows that isn't used with IIS.

**macOS**

ASP.NET Core provides the Kestrel cross-platform server implementation. In ASP.NET Core 2.0 or later, Kestrel can run as a public-facing edge server exposed directly to the Internet. Kestrel is often run in a reverse proxy configuration with Nginx or Apache.

**Linux**

ASP.NET Core provides the Kestrel cross-platform server implementation. In ASP.NET Core 2.0 or later, Kestrel can run as a public-facing edge server exposed directly to the Internet. Kestrel is often run in a reverse proxy configuration with Nginx or Apache.

### Configuration

ASP.NET Core provides a configuration framework that gets settings as name-value pairs from an ordered set of configuration providers. Built-in configuration providers are available for a variety of sources, such as .json files, .xml files, environment variables, and command-line arguments. Write custom configuration providers to support other sources.

By default, ASP.NET Core apps are configured to read from appsettings.json, environment variables, the command line, and more. When the app's configuration is loaded, values from environment variables override values from appsettings.json.

For managing confidential configuration data such as passwords, .NET Core provides the Secret Manager. For production secrets, we recommend Azure Key Vault.

### Environments

Execution environments, such as Development, Staging, and Production, are available in ASP.NET Core. Specify the environment an app is running in by setting the ASPNETCORE\_ENVIRONMENT environment variable. ASP.NET Core reads that environment variable at app startup and stores the value in an IWebHostEnvironment implementation. This implementation is available anywhere in an app via dependency injection (DI).

The following example configures the exception handler and HTTP Strict Transport Security Protocol (HSTS) middleware when ***not*** running in the Development environment:

var builder = WebApplication.CreateBuilder(args);

// Add services to the container.

builder.Services.AddRazorPages();

builder.Services.AddControllersWithViews();

var app = builder.Build();

// Configure the HTTP request pipeline.

if (!app.Environment.IsDevelopment())

{

app.UseExceptionHandler("/Error");

app.UseHsts();

}

app.UseHttpsRedirection();

app.UseStaticFiles();

app.UseAuthorization();

app.MapGet("/hi", () => "Hello!");

app.MapDefaultControllerRoute();

app.MapRazorPages();

app.Run();

### Logging

ASP.NET Core supports a logging API that works with a variety of built-in and third-party logging providers. Available providers include:

* Console
* Debug
* Event Tracing on Windows
* Windows Event Log
* TraceSource
* Azure App Service
* Azure Application Insights

To create logs, resolve an ILogger<TCategoryName> service from dependency injection (DI) and call logging methods such as LogInformation. For example:

public class IndexModel : PageModel

{

private readonly RazorPagesMovieContext \_context;

private readonly ILogger<IndexModel> \_logger;

public IndexModel(RazorPagesMovieContext context, ILogger<IndexModel> logger)

{

\_context = context;

\_logger = logger;

}

public IList<Movie> Movie { get;set; }

public async Task OnGetAsync()

{

\_logger.LogInformation("IndexModel OnGetAsync.");

Movie = await \_context.Movie.ToListAsync();

}

}

### Routing

A route is a URL pattern that is mapped to a handler. The handler is typically a Razor page, an action method in an MVC controller, or a middleware. ASP.NET Core routing gives you control over the URLs used by your app.

The following code, generated by the ASP.NET Core web application template, calls UseRouting:

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRazorPages();

var app = builder.Build();

// Configure the HTTP request pipeline.

if (!app.Environment.IsDevelopment())

{

app.UseExceptionHandler("/Error");

app.UseHsts();

}

app.UseHttpsRedirection();

app.UseStaticFiles();

app.UseRouting();

app.UseAuthorization();

app.MapRazorPages();

app.Run();

### Error handling

ASP.NET Core has built-in features for handling errors, such as:

* A developer exception page
* Custom error pages
* Static status code pages
* Startup exception handling

### Make HTTP requests

An implementation of IHttpClientFactory is available for creating HttpClient instances. The factory:

* Provides a central location for naming and configuring logical HttpClient instances. For example, register and configure a github client for accessing GitHub. Register and configure a default client for other purposes.
* Supports registration and chaining of multiple delegating handlers to build an outgoing request middleware pipeline. This pattern is similar to ASP.NET Core's inbound middleware pipeline. The pattern provides a mechanism to manage cross-cutting concerns for HTTP requests, including caching, error handling, serialization, and logging.
* Integrates with Polly, a popular third-party library for transient fault handling.
* Manages the pooling and lifetime of underlying HttpClientHandler instances to avoid common DNS problems that occur when managing HttpClient lifetimes manually.
* Adds a configurable logging experience via ILogger for all requests sent through clients created by the factory.

### Content root

The content root is the base path for:

1. The executable hosting the app (.exe).
2. Compiled assemblies that make up the app (.dll).
3. Content files used by the app, such as:
   * Razor files (.cshtml, .razor)
   * Configuration files (.json, .xml)
   * Data files (.db)

* The Web root, typically the wwwroot folder.

During development, the content root defaults to the project's root directory. This directory is also the base path for both the app's content files and the Web root. Specify a different content root by setting its path when building the host. For more information, see Content root.

### Web root

The web root is the base path for public, static resource files, such as:

* Stylesheets (.css)
* JavaScript (.js)
* Images (.png, .jpg)

By default, static files are served only from the web root directory and its sub-directories. The web root path defaults to {content root}/wwwroot. Specify a different web root by setting its path when building the host. For more information, see Web root.

Prevent publishing files in wwwroot with the <Content> project item in the project file. The following example prevents publishing content in wwwroot/local and its sub-directories:

<ItemGroup>

<Content Update="wwwroot\local\\*\*\\*.\*" CopyToPublishDirectory="Never" />

</ItemGroup>

In Razor .cshtml files, tilde-slash (~/) points to the web root. A path beginning with ~/ is referred to as a virtual path.

## Dependency Injection

ASP.NET Core supports the dependency injection (DI) software design pattern, which is a technique for achieving Inversion of Control (IoC) between classes and their dependencies.

**Project: DependencyInjectionDemo**

Steps:

1. Show IMyDependency.cs
2. Show MyDependency.cs
3. Show Index2.cshtml and Index2.cshtml.cs
4. Show DI in Program.cs for MyDependency
5. Show \_Layouts.cshtml and change /Index to /Index2
6. Run
7. Show console windows for messages when “Home” page is loaded
8. Show MyDependencyLogger
9. Run
10. Show console windows for messages when “Home” page is loaded
11. Must show “MyDependencyLogger” messages
12. Show About.cshtml and About.cshtml.cs
13. Show DI in Program.cs for MyDependencyLogger
14. Show \_Layouts.cshtml and show /About
15. Run
16. Show console windows for messages when “About” page is loaded

## ASP.NET Core – Middleware

ASP.NET Core introduced a new concept called **Middleware.** A middleware is nothing but a component (class) which is executed on every request in ASP.NET Core application. In the classic ASP.NET, HttpHandlers and HttpModules were part of request pipeline. Middleware is similar to HttpHandlers and HttpModules where both needs to be configured and executed in each request.

Typically, there will be multiple middleware in ASP.NET Core web application. It can be either framework provided middleware, added via NuGet or your own custom middleware. We can set the order of middleware execution in the request pipeline. Each middleware adds or modifies http request and optionally passes control to the next middleware component. The following figure illustrates the execution of middleware components.

Diagram

Description automatically generated

Middlewares build the request pipeline. The following figure illustrates the ASP.NET Core request processing.

ASP.NET Core Request Processing:

A picture containing diagram

Description automatically generated

### Configure Middleware

We can configure middleware in the Configure method of the Startup class using IApplicationBuilder instance. The following example adds a single middleware using Run method which returns a string "Hello World!" on each request.

**Project: MiddlewareDemo (Demo1)**

var builder = WebApplication.CreateBuilder(args);

// Add services to the container.

builder.Services.AddRazorPages();

var app = builder.Build();

// Configure the HTTP request pipeline.

if (!app.Environment.IsDevelopment())

{

app.UseExceptionHandler("/Error");

}

:

:

app.Run(async context =>

{

await context.Response.WriteAsync("Hello world!");

});

app.Run();

In the above example, Run() is an extension method on IApplicationBuilder instance which adds a terminal middleware to the application's request pipeline. The above configured middleware returns a response with a string "Hello World!" for each request.

### Understand Run Method

We used Run extension method to add middleware. The following is the signature of the Run method:

public static void Run(this IApplicationBuilder app, RequestDelegate handler)

The Run method is an extension method on IApplicationBuilder and accepts a parameter of RequestDelegate. The RequestDelegate is a delegate method which handles the request. The following is a RequestDelegate signature.

public delegate Task RequestDelegate(HttpContext context);

As you can see above, the Run method accepts a method as a parameter whose signature should match with RequestDelegate. Therefore, the method should accept the HttpContext parameter and return Task. So, you can either specify a lambda expression or specify a function in the Run method. The lambda expression specified in the Run method above is similar to the one in the example shown below.

**Note**: This code is pre-.NET 6. The new .NET 6x template does not have the Startup class.

public class Startup

{

public Startup()

{

}

public void Configure(IApplicationBuilder app, IHostingEnvironment env)

{

app.Run(MyMiddleware);

}

private Task MyMiddleware(HttpContext context)

{

return context.Response.WriteAsync("Hello World! ");

}

}

The above MyMiddleware function is not asynchronous and so will block the thread till the time it completes the execution. So, make it asynchronous by using async and await to improve performance and scalability.

// other code removed for clarity

public void Configure(IApplicationBuilder app, IHostingEnvironment env)

{

app.Run(MyMiddleware);

}

private async Task MyMiddleware(HttpContext context)

{

await context.Response.WriteAsync("Hello World! ");

}

Thus, the above code snippet is same as the one below.

app.Run(async context => await context.Response.WriteAsync("Hello World!") );

//or

app.Run(async (context) =>

{

await context.Response.WriteAsync("Hello World!");

});

So, in this way, we can configure middleware using Run method.

### Configure Multiple Middleware (Demo2)

Mostly there will be multiple middleware components in ASP.NET Core application which will be executed sequentially. The Run method adds a terminal middleware so it cannot call next middleware as it would be the last middleware in a sequence. The following will always execute the first Run method and will never reach the second Run method.

app.Run(async (context) =>

{

await context.Response.WriteAsync("Hello World From 1st Middleware");

});

// the following will never be executed

app.Run(async (context) =>

{

await context.Response.WriteAsync("Hello World From 2nd Middleware");

});

app.Run();

To configure multiple middleware, use Use() extension method. It is similar to Run() method except that it includes next parameter to invoke next middleware in the sequence. Consider the following example.

**Example: Use() (Demo3)**

app.Use(async (context, next) =>

{

await context.Response.WriteAsync("Hello World From 1st Middleware!");

await next();

});

app.Run(async (context) =>

{

await context.Response.WriteAsync("Hello World From 2nd Middleware");

});

app.Run();

The above example will display Hello World From 1st Middleware!Hello World From 2nd Middleware! in the browser.

Thus, we can use Use() method to configure multiple middlewares in the order we like.

### Add Built-in Middleware Via NuGet

ASP.NET Core is a modular framework. We can add server side features we need in our application by installing different plug-ins via NuGet. There are many middleware plug-ins available which can be used in our application.

The followings are some built-in middleware:

| Middleware | Description | Example |
| --- | --- | --- |
| Authentication | Adds authentication support. | app.UseAuthentication();  app.UseAuthorization(); |
| CORS | Configures Cross-Origin Resource Sharing. | app.UseCors(); |
| Routing | Adds routing capabilities for MVC or web form | app.UseRouting(); |
| Session | Adds support for user session. | app.UseSession(); |
| StaticFiles | Adds support for serving static files and directory browsing. | app.UseStaticFiles(); |

### Branch the middleware pipeline (Demo4)

Map extensions are used as a convention for branching the pipeline. Map branches the request pipeline based on matches of the given request path. If the request path starts with the given path, the branch is executed.

var builder = WebApplication.CreateBuilder(args);

var app = builder.Build();

app.Map("/map1", HandleMapTest1);

app.Map("/map2", HandleMapTest2);

app.Run(async context =>

{

await context.Response.WriteAsync("<p>Hello from non-Map delegate</p>");

});

app.Run();

static void HandleMapTest1(IApplicationBuilder app)

{

app.Run(async context =>

{

await context.Response.WriteAsync("Map Test 1");

});

}

static void HandleMapTest2(IApplicationBuilder app)

{

app.Run(async context =>

{

await context.Response.WriteAsync("Map Test 2");

});

}

The following table shows the requests and responses from http://localhost:1234 using the preceding code.

| **BRANCH THE MIDDLEWARE PIPELINE** | | |
| --- | --- | --- |
| **Request** | **Response** |
| localhost:1234 | Hello from non-Map delegate. |
| localhost:1234/map1 | Map Test 1 |
| localhost:1234/map2 | Map Test 2 |
| localhost:1234/map3 | Hello from non-Map delegate. |

# MVC Pattern

ASP.NET Core MVC is a rich framework for building web apps and APIs using the Model-View-Controller design pattern.

The Model-View-Controller (MVC) architectural pattern separates an application into three main groups of components: Models, Views, and Controllers. This pattern helps to achieve separation of concerns. Using this pattern, user requests are routed to a Controller which is responsible for working with the Model to perform user actions and/or retrieve results of queries. The Controller chooses the View to display to the user, and provides it with any Model data it requires.

The following diagram shows the three main components and which ones reference the others:



This delineation of responsibilities helps you scale the application in terms of complexity because it's easier to code, debug, and test something (model, view, or controller) that has a single job. It's more difficult to update, test, and debug code that has dependencies spread across two or more of these three areas. For example, user interface logic tends to change more frequently than business logic. If presentation code and business logic are combined in a single object, an object containing business logic must be modified every time the user interface is changed. This often introduces errors and requires the retesting of business logic after every minimal user interface change.

**Note**: Both the view and the controller depend on the model. However, the model depends on neither the view nor the controller. This is one of the key benefits of the separation. This separation allows the model to be built and tested independent of the visual presentation.

## Model Responsibilities

The Model in an MVC application represents the state of the application and any business logic or operations that should be performed by it. Business logic should be encapsulated in the model, along with any implementation logic for persisting the state of the application. Strongly-typed views typically use ViewModel types designed to contain the data to display on that view. The controller creates and populates these ViewModel instances from the model.

## View Responsibilities

Views are responsible for presenting content through the user interface. They use the Razor view engine to embed .NET code in HTML markup. There should be minimal logic within views, and any logic in them should relate to presenting content. If you find the need to perform a great deal of logic in view files in order to display data from a complex model, consider using a View Component, ViewModel, or view template to simplify the view.

## Controller Responsibilities

Controllers are the components that handle user interaction, work with the model, and ultimately select a view to render. In an MVC application, the view only displays information; the controller handles and responds to user input and interaction. In the MVC pattern, the controller is the initial entry point, and is responsible for selecting which model types to work with and which view to render (hence its name - it controls how the app responds to a given request).

**Note**: Controllers shouldn't be overly complicated by too many responsibilities. To keep controller logic from becoming overly complex, push business logic out of the controller and into the domain model.

**Tip**: If you find that your controller actions frequently perform the same kinds of actions, move these common actions into **filters**.

# ASP.NET Controllers

**Project: WebApplication1**

Controllers, actions, and action results are a fundamental part of how developers build apps using ASP.NET Core MVC.

A controller is used to define and group a set of actions. An action (or action method) is a method on a controller which handles requests. Controllers logically group similar actions together. This aggregation of actions allows common sets of rules, such as routing, caching, and authorization, to be applied collectively. Requests are mapped to actions through routing.

By convention, controller classes:

* Reside in the project's root-level Controllers folder.
* Inherit from Microsoft.AspNetCore.Mvc.Controller.

A controller is an instantiable class, usually public, in which at least one of the following conditions is true:

* The class name is suffixed with Controller.
* The class inherits from a class whose name is suffixed with Controller.
* The [Controller] attribute is applied to the class.

A controller class must not have an associated [NonController] attribute.

Controllers should follow the Explicit Dependencies Principle. There are a couple of approaches to implementing this principle. If multiple controller actions require the same service, consider using constructor injection to request those dependencies. If the service is needed by only a single action method, consider using Action Injection to request the dependency.

Within the **M**odel-**V**iew-**C**ontroller pattern, a controller is responsible for the initial processing of the request and instantiation of the model. Generally, business decisions should be performed within the model.

The controller takes the result of the model's processing (if any) and returns either the proper view and its associated view data or the result of the API call.

The controller is a UI-level abstraction. Its responsibilities are to ensure request data is valid and to choose which view (or result for an API) should be returned. In well-factored apps, it doesn't directly include data access or business logic. Instead, the controller delegates to services handling these responsibilities.

## Defining Actions

Public methods on a controller, except those with the [NonAction] attribute, are actions. Parameters on actions are bound to request data and are validated using model binding. Model validation occurs for everything that's model-bound. The ModelState.IsValid property value indicates whether model binding and validation succeeded.

Action methods should contain logic for mapping a request to a business concern. Business concerns should typically be represented as services that the controller accesses through dependency injection. Actions then map the result of the business action to an application state.

Actions can return anything, but frequently return an instance of IActionResult (or Task<IActionResult> for async methods) that produces a response. The action method is responsible for choosing what kind of response. The action result does the responding.

## Controller Helper Methods

Controllers usually inherit from Controller, although this isn't required. Deriving from Controller provides access to three categories of helper methods:

### 1. Methods resulting in an empty response body

No Content-Type HTTP response header is included, since the response body lacks content to describe.

There are two result types within this category: Redirect and HTTP Status Code.

* **HTTP Status Code**

This type returns an HTTP status code. A couple of helper methods of this type are BadRequest, NotFound, and Ok. For example, return BadRequest(); produces a 400 status code when executed. When methods such as BadRequest, NotFound, and Ok are overloaded, they no longer qualify as HTTP Status Code responders, since content negotiation is taking place.

* **Redirect**

This type returns a redirect to an action or destination (using Redirect, LocalRedirect, RedirectToAction, or RedirectToRoute). For example, return RedirectToAction("Complete", new {id = 123}); redirects to Complete, passing an anonymous object.

The Redirect result type differs from the HTTP Status Code type primarily in the addition of a Location HTTP response header.

### 2. Methods resulting in a non-empty response body with a predefined content type

Most helper methods in this category include a ContentType property, allowing you to set the Content-Type response header to describe the response body.

There are two result types within this category: View and Formatted Response.

* **View**

This type returns a view which uses a model to render HTML. For example, return View(customer); passes a model to the view for data-binding.

* **Formatted Response**

This type returns JSON or a similar data exchange format to represent an object in a specific manner. For example, return Json(customer); serializes the provided object into JSON format.

Other common methods of this type include File and PhysicalFile. For example, return PhysicalFile(customerFilePath, "text/xml"); returns PhysicalFileResult.

### 3. Methods resulting in a non-empty response body formatted in a content type negotiated with the client

This category is better known as **Content Negotiation**. Content negotiation applies whenever an action returns an ObjectResult type or something other than an IActionResult implementation. An action that returns a non-IActionResult implementation (for example, object) also returns a Formatted Response.

Some helper methods of this type include BadRequest, CreatedAtRoute, and Ok. Examples of these methods include return BadRequest(modelState);, return CreatedAtRoute("routename", values, newobject);, and return Ok(value);, respectively. Note that BadRequest and Ok perform content negotiation only when passed a value; without being passed a value, they instead serve as HTTP Status Code result types. The CreatedAtRoute method, on the other hand, always performs content negotiation since its overloads all require that a value be passed.

## Routing to controller actions in ASP.NET Core

ASP.NET Core controllers use the Routing middleware to match the URLs of incoming requests and map them to actions. Route templates:

* Are defined at startup in Program.cs or in attributes.
* Describe how URL paths are matched to actions.
* Are used to generate URLs for links. The generated links are typically returned in responses.

Actions are either conventionally-routed or attribute-routed. Placing a route on the controller or action makes it attribute-routed.

### Set up conventional route

The ASP.NET Core MVC template generates conventional routing code similar to the following:

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddControllersWithViews();

var app = builder.Build();

if (!app.Environment.IsDevelopment())

{

app.UseExceptionHandler("/Home/Error");

app.UseHsts();

}

app.UseHttpsRedirection();

app.UseStaticFiles();

app.UseRouting();

app.UseAuthorization();

app.MapControllerRoute(

name: "default",

pattern: "{controller=Home}/{action=Index}/{id?}");

app.Run();

MapControllerRoute is used to create a single route. The single route is named default route. Most apps with controllers and views use a route template similar to the default route. REST APIs should use attribute routing.

The route template "{controller=Home}/{action=Index}/{id?}":

* Matches a URL path like /Products/Details/5
* Extracts the route values { controller = Products, action = Details, id = 5 } by tokenizing the path. The extraction of route values results in a match if the app has a controller named ProductsController and a Details action:

public class ProductsController : Controller

{

public IActionResult Details(int id)

{

return ControllerContext.MyDisplayRouteInfo(id);

}

}

* /Products/Details/5 model binds the value of id = 5 to set the id parameter to 5.
* {controller=Home} defines Home as the default controller.
* {action=Index} defines Index as the default action.
* The ? character in {id?} defines id as optional.
* Default and optional route parameters don't need to be present in the URL path for a match
* Matches the URL path /.
* Produces the route values { controller = Home, action = Index }.

The values for controller and action make use of the default values. id doesn't produce a value since there's no corresponding segment in the URL path. / only matches if there exists a HomeController and Index action:

public class HomeController : Controller

{

public IActionResult Index() { ... }

}

Using the preceding controller definition and route template, the HomeController.Index action is run for the following URL paths:

* /Home/Index/17
* /Home/Index
* /Home
* /

The URL path / uses the route template default Home controllers and Index action. The URL path /Home uses the route template default Index action.

### Multiple conventional routes

Multiple conventional routes can be added inside UseEndpoints by adding more calls to MapControllerRoute and MapAreaControllerRoute. Doing so allows defining multiple conventions, or to adding conventional routes that are dedicated to a specific action, such as:

app.MapControllerRoute(name: "blog",

pattern: "blog/{\*article}",

defaults: new { controller = "Blog", action = "Article" });

app.MapControllerRoute(name: "default",

pattern: "{controller=Home}/{action=Index}/{id?}");

The blog route in the preceding code is a **dedicated conventional route**. It's called a dedicated conventional route because:

* It uses conventional routing.
* It's dedicated to a specific action.

Because controller and action don't appear in the route template "blog/{\*article}" as parameters:

* They can only have the default values { controller = "Blog", action = "Article" }.
* This route always maps to the action BlogController.Article.

/Blog, /Blog/Article, and /Blog/{any-string} are the only URL paths that match the blog route.

The preceding example:

* blog route has a higher priority for matches than the default route because it is added first.
* Is an example of Slug style routing where it's typical to have an article name as part of the URL.

# ASP.NET - Model Binding

**Project: ModelBindingWebApi**

**Controller**: PersonController

[BindingProperty] – At property level.

[BindingProperties] – At class level.

**Project: WebApplication1**

**Controller**: ModelBindingController

[BindingProperty] – At property level.

[BindingProperties] – At class level.

# ASP.NET – Filters

*Filters* in ASP.NET Core allow code to run before or after specific stages in the request processing pipeline.

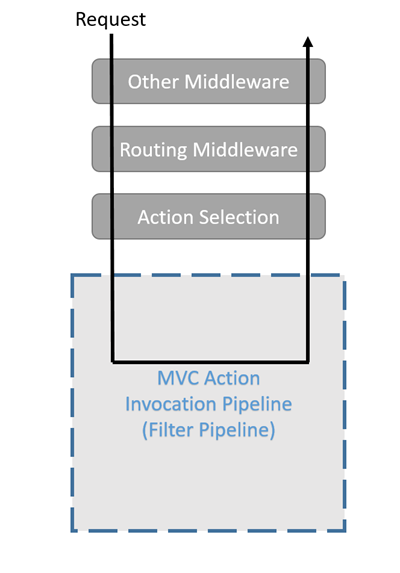
Built-in filters handle tasks such as:

* Authorization, preventing access to resources a user isn't authorized for.
* Response caching, short-circuiting the request pipeline to return a cached response.

Custom filters can be created to handle cross-cutting concerns. Examples of cross-cutting concerns include error handling, caching, configuration, authorization, and logging. Filters avoid duplicating code. For example, an error handling exception filter could consolidate error handling.

## How filters work

Filters run within the ASP.NET Core action invocation pipeline, sometimes referred to as the filter pipeline. The filter pipeline runs after ASP.NET Core selects the action to execute:

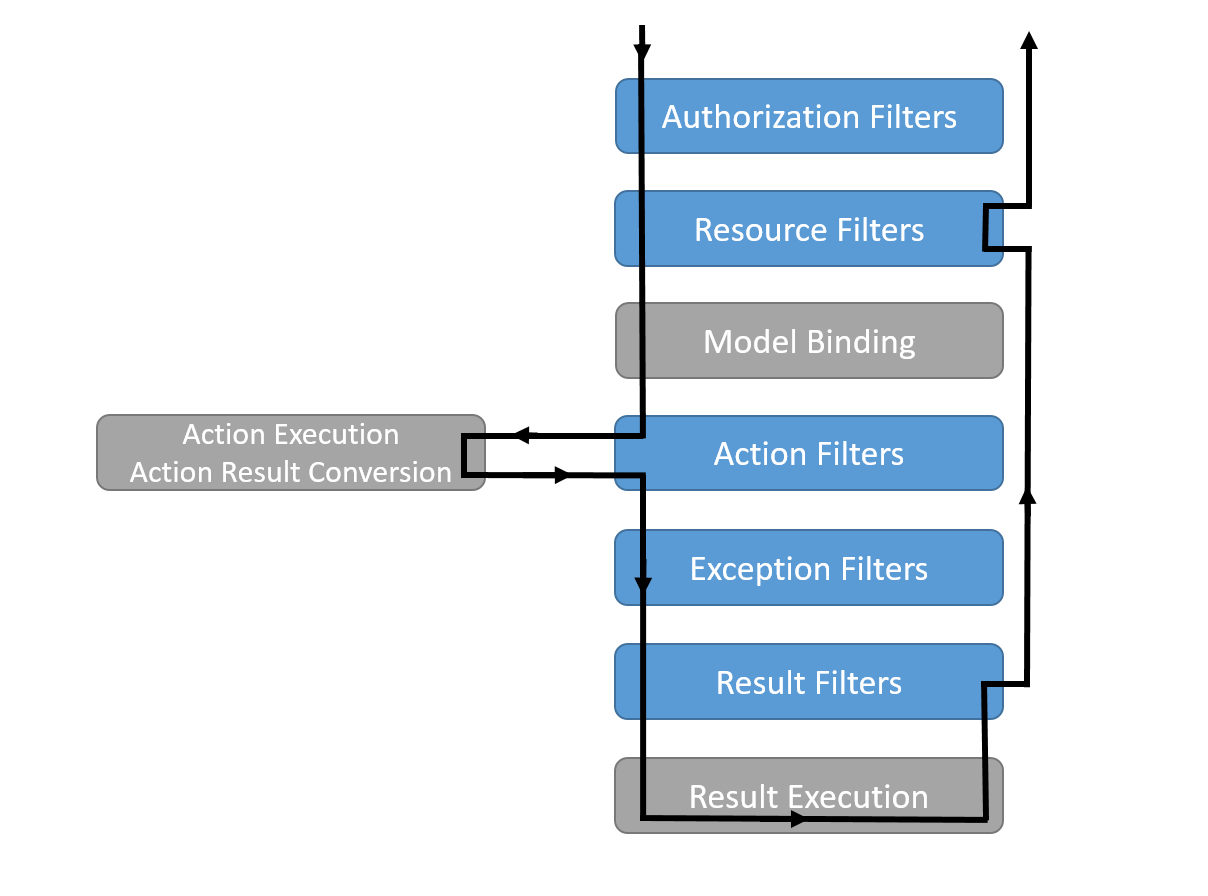


## Filter types

Each filter type is executed at a different stage in the filter pipeline:

* Authorization filters:
  + Run first.
  + Determine whether the user is authorized for the request.
  + Short-circuit the pipeline if the request is not authorized.
* Resource filters:
  + Run after authorization.
  + OnResourceExecuting runs code before the rest of the filter pipeline. For example, OnResourceExecuting runs code before model binding.
  + OnResourceExecuted runs code after the rest of the pipeline has completed.
* Action filters:
  + Run immediately before and after an action method is called.
  + Can change the arguments passed into an action.
  + Can change the result returned from the action.
  + Are **not** supported in Razor Pages.
* Exception filters apply global policies to unhandled exceptions that occur before the response body has been written to.
* Result filters:
  + Run immediately before and after the execution of action results.
  + Run only when the action method executes successfully.
  + Are useful for logic that must surround view or formatter execution.

The following diagram shows how filter types interact in the filter pipeline:



## Action Filter Attribute

**Project**: FilterDemos

**Code**: FilterController.cs , SampleActionFilter.cs and SampleActionFilterAsync.cs

***Note****: Show both, Action scope and Controller scope.*

**SampleActionFilter.cs**

using Microsoft.AspNetCore.Mvc.Filters;

namespace FilterDemos

{

public class SampleActionFilter : ActionFilterAttribute, IActionFilter

{

public override void OnActionExecuted(ActionExecutedContext context)

{

Console.WriteLine("SampleActionFilter.OnActionExecuted...");

Console.WriteLine(string.Format("Action Method {0} executing at {1}",

context.ActionDescriptor.DisplayName,

DateTime.Now.ToShortDateString())

);

}

public override void OnActionExecuting(ActionExecutingContext context)

{

Console.WriteLine("SampleActionFilter.OnActionExecuting...");

Console.WriteLine(string.Format("Action Method {0} executing at {1}",

context.ActionDescriptor.DisplayName,

DateTime.Now.ToShortDateString())

);

}

}

}

**SampleActionFilterAsync.cs**

**FilterController.cs**

using Microsoft.AspNetCore.Http;

using Microsoft.AspNetCore.Mvc;

using FilterDemos;

using FilterDemos.Models;

namespace FilterDemos.Controllers

{

[Route("api/[controller]")]

[ApiController]

public class FilterController : ControllerBase

{

[SampleActionFilter]

public async Task<ActionResult<string>> Get()

{

Console.WriteLine("FilterController.Get()...");

return "FilterController.Get()...";

}

[SampleActionFilterAsync]

[Route("Get2")]

public async Task<ActionResult<string>> Get2()

{

Console.WriteLine("FilterController.Get2()...");

return "FilterController.Get2()...";

}

}

}

### The Scope of Action Filters

Like the other types of filters, the action filter can be added to different scope levels: Global, Action, Controller.

If we want to use our filter globally, we need to register it inside the AddControllers() method in Program.cs

builder.Services.AddControllers(config =>

{

config.Filters.Add(new SampleActionFilter());

}

);

After this, the filter will apply to all Actions of all Controllers, without having to apply the attribute to them explicitly. There can be other action filters that can be applied to specific Controllers and Actions.

### Order of Execution

Order of execution of filters - Action Filters

### Changing the Order

**Project**: FilterDemos

**Code**: FilterController.cs

[SampleActionFilter(Order = 2)]

[SampleActionFilterAsync(Order = 1)]

public async Task<ActionResult<string>> Get()

{

Console.WriteLine("FilterController.Get()...");

return "FilterController.Get()...";

}

### Adding Validations

**Project**: FilterDemos

**Code**: FilterController, Person.cs and ValidationFilterAttribute.cs

**IEntity.cs**

namespace FilterDemos.Models

{

public interface IEntity

{

int Id { get; set; }

}

}

**Person.cs**

namespace FilterDemos.Models

{

public class Person : IEntity

{

public int Id { get ; set ; }

public string Name { get; set; } = string.Empty;

}

public class InvalidPerson

{

public int Id { get; set; }

public string Name { get; set; } = string.Empty;

}

}

**ValidationFilterAttribute.cs**

using FilterDemos.Models;

using Microsoft.AspNetCore.Mvc;

using Microsoft.AspNetCore.Mvc.Filters;

namespace FilterDemos

{

public class ValidationFilterAttribute : ActionFilterAttribute, IActionFilter

{

public override void OnActionExecuted(ActionExecutedContext context)

{

Console.WriteLine("ValidationFilterAttribute.OnActionExecuted...");

Console.WriteLine(string.Format("Action Method {0} executing at {1}",

context.ActionDescriptor.DisplayName,

DateTime.Now.ToShortDateString())

);

}

public override void OnActionExecuting(ActionExecutingContext context)

{

Console.WriteLine("ValidationFilterAttribute.OnActionExecuting...");

Console.WriteLine(string.Format("Action Method {0} executing at {1}",

context.ActionDescriptor.DisplayName,

DateTime.Now.ToShortDateString())

);

var param = context.ActionArguments.SingleOrDefault(p => p.Value is IEntity);

if (param.Value == null)

{

context.Result = new BadRequestObjectResult("Object is null");

return;

}

}

}

}

**FilterController.cs**

using Microsoft.AspNetCore.Http;

using Microsoft.AspNetCore.Mvc;

using FilterDemos;

using FilterDemos.Models;

namespace FilterDemos.Controllers

{

[Route("api/[controller]")]

[ApiController]

public class FilterController : ControllerBase

{

:

:

[HttpPost]

[ValidationFilter]

public async Task<ActionResult<string>> PostPerson([FromBody] InvalidPerson person)

{

return person.Name;

}

}

}

## Service Filter Attribute

Service filter implementation types are registered in Program.cs. A ServiceFilterAttribute retrieves an instance of the filter from DI.

The following code shows the LoggingResponseHeaderFilterService class, which uses DI:

**LoggingResponseHeaderFilterService.cs**

using Microsoft.AspNetCore.Mvc.Filters;

namespace FilterDemos

{

public class LoggingResponseHeaderFilterService : IResultFilter

{

private readonly ILogger \_logger;

public LoggingResponseHeaderFilterService(

ILogger<LoggingResponseHeaderFilterService> logger) =>

\_logger = logger;

public void OnResultExecuting(ResultExecutingContext context)

{

\_logger.LogInformation(

$"- {nameof(LoggingResponseHeaderFilterService)}.{nameof(OnResultExecuting)}");

context.HttpContext.Response.Headers.Add(

nameof(OnResultExecuting), nameof(LoggingResponseHeaderFilterService));

}

public void OnResultExecuted(ResultExecutedContext context)

{

\_logger.LogInformation(

$"- {nameof(LoggingResponseHeaderFilterService)}.{nameof(OnResultExecuted)}");

}

}

}

In the following code, LoggingResponseHeaderFilterService is added to the DI container in **Program.cs**:

builder.Services.AddScoped<LoggingResponseHeaderFilterService>();

In the following code, the ServiceFilter attribute retrieves an instance of the LoggingResponseHeaderFilterService filter from DI in **FilterController.cs**:

[ServiceFilter(typeof(LoggingResponseHeaderFilterService))]

[HttpGet]

[Route("WithServiceFilter")]

public IActionResult WithServiceFilter() =>

Content($"- {nameof(FilterController)}.{nameof(WithServiceFilter)}");

# Attribute routing for REST APIs

REST APIs should use attribute routing to model the app's functionality as a set of resources where operations are represented by HTTP verbs.

Attribute routing uses a set of attributes to map actions directly to route templates. The following code is typical for a REST API and is used in the next sample:

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddControllers();

var app = builder.Build();

app.UseHttpsRedirection();

app.UseAuthorization();

app.MapControllers();

app.Run();

In the preceding code, MapControllers is called inside UseEndpoints to map attribute routed controllers.

In the following example:

* HomeController matches a set of URLs similar to what the default conventional route {controller=Home}/{action=Index}/{id?} matches.

public class HomeController : Controller

{

[Route("")]

[Route("Home")]

[Route("Home/Index")]

[Route("Home/Index/{id?}")]

public IActionResult Index(int? id)

{

return ControllerContext.MyDisplayRouteInfo(id);

}

[Route("Home/About")]

[Route("Home/About/{id?}")]

public IActionResult About(int? id)

{

return ControllerContext.MyDisplayRouteInfo(id);

}

}

The HomeController.Index action is run for any of the URL paths /, /Home, /Home/Index, or /Home/Index/3.

This example highlights a key programming difference between attribute routing and conventional routing. Attribute routing requires more input to specify a route. The conventional default route handles routes more succinctly. However, attribute routing allows and requires precise control of which route templates apply to each action.

With attribute routing, the controller and action names play no part in which action is matched, unless token replacement is used. The following example matches the same URLs as the previous example:

public class MyDemoController : Controller

{

[Route("")]

[Route("Home")]

[Route("Home/Index")]

[Route("Home/Index/{id?}")]

public IActionResult MyIndex(int? id)

{

return ControllerContext.MyDisplayRouteInfo(id);

}

[Route("Home/About")]

[Route("Home/About/{id?}")]

public IActionResult MyAbout(int? id)

{

return ControllerContext.MyDisplayRouteInfo(id);

}

}

The following code uses token replacement for action and controller:

public class HomeController : Controller

{

[Route("")]

[Route("Home")]

[Route("[controller]/[action]")]

public IActionResult Index()

{

return ControllerContext.MyDisplayRouteInfo();

}

[Route("[controller]/[action]")]

public IActionResult About()

{

return ControllerContext.MyDisplayRouteInfo();

}

}

The following code applies [Route("[controller]/[action]")] to the controller:

[Route("[controller]/[action]")]

public class HomeController : Controller

{

[Route("~/")]

[Route("/Home")]

[Route("~/Home/Index")]

public IActionResult Index()

{

return ControllerContext.MyDisplayRouteInfo();

}

public IActionResult About()

{

return ControllerContext.MyDisplayRouteInfo();

}

}

In the preceding code, the Index method templates must prepend / or ~/ to the route templates. Route templates applied to an action that begin with / or ~/ don't get combined with route templates applied to the controller.