

# Contents

## Programming Concepts

Assemblies and the Global Assembly Cache

Asynchronous Programming with `async` and `await`

Attributes

Caller Information

Collections

Covariance and Contravariance

Expression Trees

Iterators

Language-Integrated Query (LINQ)

Object-Oriented Programming

Reflection

Serialization (C#)

# Programming Concepts (C#)

11/9/2018 • 2 minutes to read • [Edit Online](#)

This section explains programming concepts in the C# language.

## In This Section

TITLE	DESCRIPTION
<a href="#">Assemblies and the Global Assembly Cache (C#)</a>	Describes how to create and use assemblies.
<a href="#">Asynchronous Programming with async and await (C#)</a>	Describes how to write asynchronous solutions by using the <a href="#">async</a> and <a href="#">await</a> keywords in C#. Includes a walkthrough.
<a href="#">Attributes (C#)</a>	Discusses how to provide additional information about programming elements such as types, fields, methods, and properties by using attributes.
<a href="#">Caller Information (C#)</a>	Describes how to obtain information about the caller of a method. This information includes the file path and the line number of the source code and the member name of the caller.
<a href="#">Collections (C#)</a>	Describes some of the types of collections provided by the .NET Framework. Demonstrates how to use simple collections and collections of key/value pairs.
<a href="#">Covariance and Contravariance (C#)</a>	Shows how to enable implicit conversion of generic type parameters in interfaces and delegates.
<a href="#">Expression Trees (C#)</a>	Explains how you can use expression trees to enable dynamic modification of executable code.
<a href="#">Iterators (C#)</a>	Describes iterators, which are used to step through collections and return elements one at a time.
<a href="#">Language-Integrated Query (LINQ) (C#)</a>	Discusses the powerful query capabilities in the language syntax of C#, and the model for querying relational databases, XML documents, datasets, and in-memory collections.
<a href="#">Object-Oriented Programming (C#)</a>	Describes common object-oriented concepts, including encapsulation, inheritance, and polymorphism.
<a href="#">Reflection (C#)</a>	Explains how to use reflection to 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.
<a href="#">Serialization (C#)</a>	Describes key concepts in binary, XML, and SOAP serialization.

## Related Sections

Performance Tips	Discusses several basic rules that may help you increase the performance of your application.

# Assemblies and the Global Assembly Cache (C#)

1/23/2019 • 3 minutes to read • [Edit Online](#)

Assemblies form the fundamental unit of deployment, version control, reuse, activation scoping, and security permissions for a .NET-based application. Assemblies take the form of an executable (.exe) file or dynamic link library (.dll) file, and are the building blocks of the .NET Framework. They provide the common language runtime with the information it needs to be aware of type implementations. You can think of an assembly as a collection of types and resources that form a logical unit of functionality and are built to work together.

Assemblies can contain one or more modules. For example, larger projects may be planned in such a way that several individual developers work on separate modules, all coming together to create a single assembly. For more information about modules, see the topic [How to: Build a Multifile Assembly](#).

Assemblies have the following properties:

- Assemblies are implemented as .exe or .dll files.
- You can share an assembly between applications by putting it in the global assembly cache. Assemblies must be strong-named before they can be included in the global assembly cache. For more information, see [Strong-Named Assemblies](#).
- Assemblies are only loaded into memory if they are required. If they are not used, they are not loaded. This means that assemblies can be an efficient way to manage resources in larger projects.
- You can programmatically obtain information about an assembly by using reflection. For more information, see [Reflection \(C#\)](#).
- If you want to load an assembly only to inspect it, use a method such as [ReflectionOnlyLoadFrom](#).

## Assembly Manifest

Within every assembly, there is an *assembly manifest*. Similar to a table of contents, the assembly manifest contains the following:

- The assembly's identity (its name and version).
- A file table describing all the other files that make up the assembly, for example, any other assemblies you created that your .exe or .dll file relies on, or even bitmap or Readme files.
- An *assembly reference list*, which is a list of all external dependencies—.dlls or other files your application needs that may have been created by someone else. Assembly references contain references to both global and private objects. Global objects reside in the global assembly cache, an area available to other applications. Private objects must be in a directory at either the same level as or below the directory in which your application is installed.

Because assemblies contain information about content, versioning, and dependencies, the applications you create with C# do not rely on Windows registry values to function properly. Assemblies reduce .dll conflicts and make your applications more reliable and easier to deploy. In many cases, you can install a .NET-based application simply by copying its files to the target computer.

For more information see [Assembly Manifest](#).

## Adding a Reference to an Assembly

To use an assembly, you must add a reference to it. Next, you use the [using directive](#) to choose the namespace of the items you want to use. Once an assembly is referenced and imported, all the accessible classes, properties, methods, and other members of its namespaces are available to your application as if their code were part of your source file.

In C#, you can also use two versions of the same assembly in a single application. For more information, see [extern alias](#).

## Creating an Assembly

Compile your application by clicking **Build** on the **Build** menu or by building it from the command line using the command-line compiler. For details about building assemblies from the command line, see [Command-line Building With csc.exe](#).

### NOTE

To build an assembly in Visual Studio, on the **Build** menu choose **Build**.

## See also

- [C# Programming Guide](#)
- [Assemblies in the Common Language Runtime](#)
- [Friend Assemblies \(C#\)](#)
- [How to: Share an Assembly with Other Applications \(C#\)](#)
- [How to: Load and Unload Assemblies \(C#\)](#)
- [How to: Determine If a File Is an Assembly \(C#\)](#)
- [How to: Create and Use Assemblies Using the Command Line \(C#\)](#)
- [Walkthrough: Embedding Types from Managed Assemblies in Visual Studio \(C#\)](#)
- [Walkthrough: Embedding Type Information from Microsoft Office Assemblies in Visual Studio \(C#\)](#)

# Asynchronous programming with async and await (C#)

2/6/2019 • 16 minutes to read • [Edit Online](#)

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.

This topic provides an overview of when and how to use async programming and includes links to support topics that contain details and examples.

## 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.

APPLICATION AREA	.NET TYPES WITH ASYNC METHODS	WINDOWS RUNTIME TYPES WITH ASYNC METHODS
Web access	<a href="#">HttpClient</a>	<a href="#">SyndicationClient</a>
Working with files	<a href="#">StreamWriter</a> , <a href="#">StreamReader</a> , <a href="#">XmlReader</a>	<a href="#">StorageFile</a>
Working with images		<a href="#">MediaCapture</a> , <a href="#">BitmapEncoder</a> , <a href="#">BitmapDecoder</a>
WCF programming	<a href="#">Synchronous and Asynchronous Operations</a>	

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 easier 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 the .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 completely familiar to you. The comments call out the features that you add to create the asynchrony.

You can find a complete Windows Presentation Foundation (WPF) example file at the end of this topic, and you can download the sample from [Async Sample: Example from "Asynchronous Programming with Async and Await"](#).

```
// Three things to note in the signature:
// - The method has an async modifier.
// - The return type is Task or Task<T>. (See "Return Types" section.)
//   Here, it is Task<int> because the return statement returns an integer.
// - The method name ends in "Async."
async Task<int> AccessTheWebAsync()
{
    // You need to add a reference to System.Net.Http to declare client.
    using (HttpClient client = new HttpClient())
    {
        // GetStringAsync returns a Task<string>. That means that when you await the
        // task you'll get a string (urlContents).
        Task<string> getStringTask = client.GetStringAsync("https://docs.microsoft.com");

        // You can do work here that doesn't rely on the string from GetStringAsync.
        DoIndependentWork();

        // The await operator suspends AccessTheWebAsync.
        // - AccessTheWebAsync can't continue until getStringTask is complete.
        // - Meanwhile, control returns to the caller of AccessTheWebAsync.
        // - Control resumes here when getStringTask is complete.
        // - The await operator then retrieves the string result from getStringTask.
        string urlContents = await getStringTask;

        // The return statement specifies an integer result.
        // Any methods that are awaiting AccessTheWebAsync retrieve the length value.
        return urlContents.Length;
    }
}
```

If `AccessTheWebAsync` 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 urlContents = await client.GetStringAsync("https://docs.microsoft.com");
```

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).

For more information, see the [Return Types and Parameters](#) section.

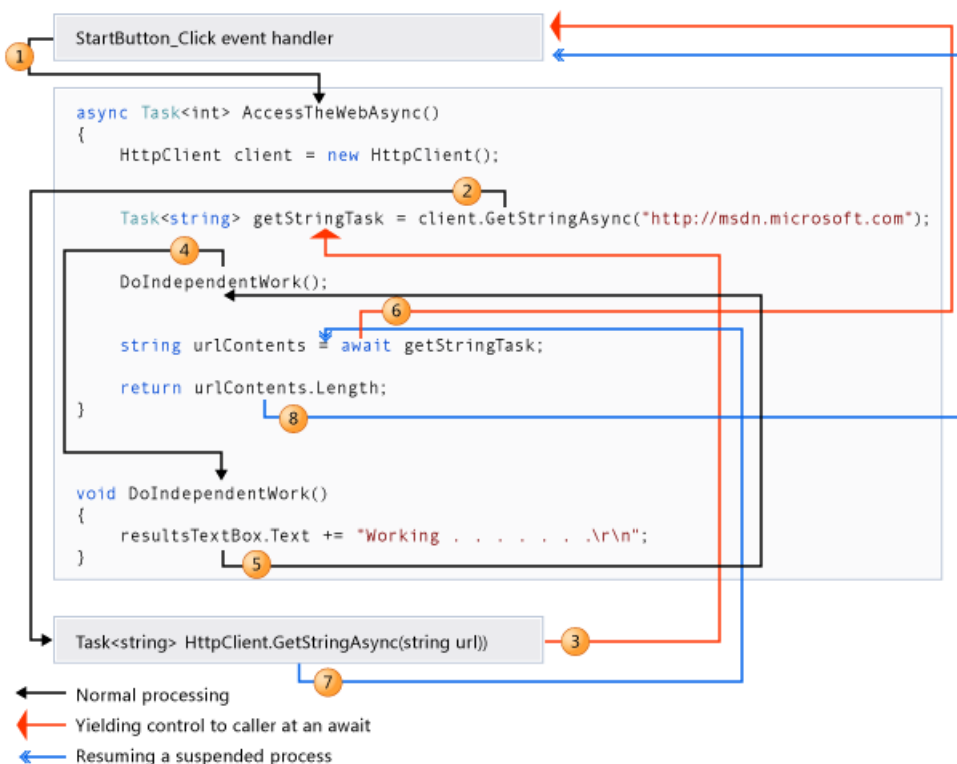
- The method usually includes at least one await expression, which marks a point where the method can't continue until the awaited asynchronous operation is complete. In the meantime, the method is suspended, and control returns to the method's caller. The next section of this topic illustrates what happens at the suspension point.

In async methods, you use the provided keywords and types to indicate what you want to do, and the compiler does the rest, including keeping track of what must happen when control returns to an await point in a suspended method. Some routine processes, such as loops and exception handling, can be difficult to handle in traditional asynchronous code. In an async method, you write these elements much as you would in a synchronous solution, and the problem is solved.

For more information about asynchrony in previous versions of the .NET Framework, see [TPL and Traditional .NET Framework Asynchronous Programming](#).

## 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.

- An event handler calls and awaits the `AccessTheWebAsync` async method.
- `AccessTheWebAsync` creates an `HttpClient` instance and calls the `GetStringAsync` asynchronous method to download the contents of a website as a string.
- 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, `AccessTheWebAsync`.

`GetStringAsync` returns a `Task<TResult>` where `TResult` is a string, and `AccessTheWebAsync` 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.

4. Because `getStringTask` hasn't been awaited yet, `AccessTheWebAsync` 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`.
5. `DoIndependentWork` is a synchronous method that does its work and returns to its caller.
6. `AccessTheWebAsync` has run out of work that it can do without a result from `getStringTask`. `AccessTheWebAsync` 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, `AccessTheWebAsync` uses an await operator to suspend its progress and to yield control to the method that called `AccessTheWebAsync`. `AccessTheWebAsync` 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`) is complete before `AccessTheWebAsync` awaits it, control remains in `AccessTheWebAsync`. The expense of suspending and then returning to `AccessTheWebAsync` would be wasted if the called asynchronous process (`getStringTask`) has already completed and `AccessTheWebSync` doesn't have to wait for the final result.

Inside the caller (the event handler in this example), the processing pattern continues. The caller might do other work that doesn't depend on the result from `AccessTheWebAsync` before awaiting that result, or the caller might await immediately. The event handler is waiting for `AccessTheWebAsync`, and `AccessTheWebAsync` is waiting for `GetStringAsync`.

7. `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 `urlContents`.
8. When `AccessTheWebAsync` has the string result, the method can calculate the length of the string. Then the work of `AccessTheWebAsync` 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.

For more information about control flow, see [Control Flow in Async Programs \(C#\)](#).

## API async methods

You might be wondering where to find methods such as `GetStringAsync` that support async programming. The .NET Framework 4.5 or higher and .NET Core contain many members that work with `async` and `await`. You can recognize them by the "Async" suffix that's appended to the member name, and by their return type of `Task` or `Task<TResult>`. For example, the `System.IO.Stream` class contains methods such as `CopyToAsync`, `ReadAsync`, and `WriteAsync` alongside the synchronous methods `CopyTo`, `Read`, and `Write`.

The Windows Runtime also contains many methods that you can use with `async` and `await` in Windows apps. For more information, see [Threading and async programming](#) for UWP development, and [Asynchronous programming \(Windows Store apps\)](#) and [Quickstart: Calling asynchronous APIs in C# or Visual Basic](#) if you use earlier versions of the Windows Runtime.

## Threads

Async methods are intended to be non-blocking operations. An `await` expression in an async method doesn't block the current thread while the awaited task is running. Instead, the expression signs up the rest of the method as a continuation and returns control to the caller of the async method.

The `async` and `await` keywords don't cause additional threads to be created. Async methods don't require multithreading because an async method doesn't run on its own thread. The method runs on the current synchronization context and uses time on the thread only when the method is active. You can use [Task.Run](#) to move CPU-bound work to a background thread, but a background thread doesn't help with a process that's just waiting for results to become available.

The async-based approach to asynchronous programming is preferable to existing approaches in almost every case. In particular, this approach is better than the [BackgroundWorker](#) class for I/O-bound operations because the code is simpler and you don't have to guard against race conditions. In combination with the [Task.Run](#) method, async programming is better than [BackgroundWorker](#) for CPU-bound operations because async programming separates the coordination details of running your code from the work that `Task.Run` transfers to the threadpool.

## async and await

If you specify that a method is an async method by using the `async` modifier, you enable the following two capabilities.

- The marked async method can use [await](#) to designate suspension points. The `await` operator tells the compiler that the async method can't continue past that point until the awaited asynchronous process is complete. In the meantime, control returns to the caller of the async method.

The suspension of an async method at an `await` expression doesn't constitute an exit from the method, and `finally` blocks don't run.

- The marked async method can itself be awaited by methods that call it.

An async method typically contains one or more occurrences of an `await` operator, but the absence of `await` expressions doesn't cause a compiler error. If an async method doesn't use an `await` operator to mark a suspension point, the method executes as a synchronous method does, despite the `async` modifier. The compiler issues a warning for such methods.

`async` and `await` are contextual keywords. For more information and examples, see the following topics:

- [async](#)
- [await](#)

## Return types and parameters

An async method typically returns a [Task](#) or a [Task<TResult>](#). Inside an async method, an `await` operator is applied to a task that's returned from a call to another async method.

You specify [Task<TResult>](#) as the return type if the method contains a [return](#) statement that specifies an operand of type `TResult`.

You use [Task](#) as the return type if the method has no return statement or has a return statement that doesn't return

an operand.

Starting with C# 7.0, you can also specify any other return type, provided that the type includes a `GetAwaiter` method. `ValueTask<TResult>` is an example of such a type. It is available in the `System.Threading.Tasks.Extensions` NuGet package.

The following example shows how you declare and call a method that returns a `Task<TResult>` or a `Task`.

```
// Signature specifies Task<TResult>
async Task<int> GetTaskOfTResultAsync()
{
    int hours = 0;
    await Task.Delay(0);
    // Return statement specifies an integer result.
    return hours;
}

// Calls to GetTaskOfTResultAsync
Task<int> returnedTaskTResult = GetTaskOfTResultAsync();
int intResult = await returnedTaskTResult;
// or, in a single statement
int intResult = await GetTaskOfTResultAsync();

// Signature specifies Task
async Task GetTaskAsync()
{
    await Task.Delay(0);
    // The method has no return statement.
}

// Calls to GetTaskAsync
Task returnedTask = GetTaskAsync();
await returnedTask;
// or, in a single statement
await GetTaskAsync();
```

Each returned task represents ongoing work. A task encapsulates information about the state of the asynchronous process and, eventually, either the final result from the process or the exception that the process raises if it doesn't succeed.

An async method can also have a `void` return type. This return type is used primarily to define event handlers, where a `void` return type is required. Async event handlers often serve as the starting point for async programs.

An async method that has a `void` return type can't be awaited, and the caller of a void-returning method can't catch any exceptions that the method throws.

An async method can't declare `in`, `ref` or `out` parameters, but the method can call methods that have such parameters. Similarly, an async method can't return a value by reference, although it can call methods with `ref` return values.

For more information and examples, see [Async Return Types \(C#\)](#). For more information about how to catch exceptions in async methods, see [try-catch](#).

Asynchronous APIs in Windows Runtime programming have one of the following return types, which are similar to tasks:

- `IAsyncOperation<TResult>`, which corresponds to `Task<TResult>`
- `IAsyncAction`, which corresponds to `Task`
- `IAsyncActionWithProgress<TProgress>`

- [IAsyncOperationWithProgress<TResult, TProgress>](#)

## Naming convention

By convention, you append "Async" to the names of methods that have an `async` modifier.

You can ignore the convention where an event, base class, or interface contract suggests a different name. For example, you shouldn't rename common event handlers, such as `Button1_Click`.

## Related topics and samples (Visual Studio)

TITLE	DESCRIPTION	SAMPLE
<a href="#">Walkthrough: Accessing the Web by Using async and await (C#)</a>	Shows how to convert a synchronous WPF solution to an asynchronous WPF solution. The application downloads a series of websites.	<a href="#">Async Sample: Accessing the Web Walkthrough</a>
<a href="#">How to: Extend the async Walkthrough by Using Task.WhenAll (C#)</a>	Adds <a href="#">Task.WhenAll</a> to the previous walkthrough. The use of <code>WhenAll</code> starts all the downloads at the same time.	
<a href="#">How to: Make Multiple Web Requests in Parallel by Using async and await (C#)</a>	Demonstrates how to start several tasks at the same time.	<a href="#">Async Sample: Make Multiple Web Requests in Parallel</a>
<a href="#">Async Return Types (C#)</a>	Illustrates the types that async methods can return and explains when each type is appropriate.	
<a href="#">Control Flow in Async Programs (C#)</a>	Traces in detail the flow of control through a succession of await expressions in an asynchronous program.	<a href="#">Async Sample: Control Flow in Async Programs</a>
<a href="#">Fine-Tuning Your Async Application (C#)</a>	Shows how to add the following functionality to your async solution: <ul style="list-style-type: none"> <li>- <a href="#">Cancel an Async Task or a List of Tasks (C#)</a></li> <li>- <a href="#">Cancel Async Tasks after a Period of Time (C#)</a></li> <li>- <a href="#">Cancel Remaining Async Tasks after One Is Complete (C#)</a></li> <li>- <a href="#">Start Multiple Async Tasks and Process Them As They Complete (C#)</a></li> </ul>	<a href="#">Async Sample: Fine Tuning Your Application</a>
<a href="#">Handling Reentrancy in Async Apps (C#)</a>	Shows how to handle cases in which an active asynchronous operation is restarted while it's running.	
<a href="#">WhenAny: Bridging between the .NET Framework and the Windows Runtime</a>	Shows how to bridge between Task types in the .NET Framework and <code>IAsyncOperations</code> in the Windows Runtime so that you can use <a href="#">WhenAny</a> with a Windows Runtime method.	<a href="#">Async Sample: Bridging between .NET and Windows Runtime (AsTask and WhenAny)</a>

TITLE	DESCRIPTION	SAMPLE
Async Cancellation: Bridging between the .NET Framework and the Windows Runtime	Shows how to bridge between Task types in the .NET Framework and IAsyncOperations in the Windows Runtime so that you can use <a href="#">CancellationTokenSource</a> with a Windows Runtime method.	<a href="#">Async Sample: Bridging between .NET and Windows Runtime (AsTask &amp; Cancellation)</a>
<a href="#">Using Async for File Access (C#)</a>	Lists and demonstrates the benefits of using async and await to access files.	
<a href="#">Task-based Asynchronous Pattern (TAP)</a>	Describes a new pattern for asynchrony in the .NET Framework. The pattern is based on the <a href="#">Task</a> and <a href="#">Task&lt;TResult&gt;</a> types.	
<a href="#">Async Videos on Channel 9</a>	Provides links to a variety of videos about async programming.	

## Complete example

The following code is the MainWindow.xaml.cs file from the Windows Presentation Foundation (WPF) application that this topic discusses. You can download the sample from [Async Sample: Example from "Asynchronous Programming with Async and Await"](#).

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Data;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Imaging;
using System.Windows.Navigation;
using System.Windows.Shapes;

// Add a using directive and a reference for System.Net.Http;
using System.Net.Http;

namespace AsyncFirstExample
{
    public partial class MainWindow : Window
    {
        // Mark the event handler with async so you can use await in it.
        private async void StartButton_Click(object sender, RoutedEventArgs e)
        {
            // Call and await separately.
            //Task<int> getLengthTask = AccessTheWebAsync();
            //// You can do independent work here.
            //int contentLength = await getLengthTask;

            int contentLength = await AccessTheWebAsync();

            resultsTextBox.Text +=
                $"{Environment.NewLine}Length of the downloaded string: {contentLength}{Environment.NewLine}";
        }
    }
}
```

```

// Three things to note in the signature:
// - The method has an async modifier.
// - The return type is Task or Task<T>. (See "Return Types" section.)
//   Here, it is Task<int> because the return statement returns an integer.
// - The method name ends in "Async."
async Task<int> AccessTheWebAsync()
{
    // You need to add a reference to System.Net.Http to declare client.
    using (HttpClient client = new HttpClient())
    {
        // GetStringAsync returns a Task<string>. That means that when you await the
        // task you'll get a string (urlContents).
        Task<string> getStringTask = client.GetStringAsync("https://docs.microsoft.com");

        // You can do work here that doesn't rely on the string from GetStringAsync.
        DoIndependentWork();

        // The await operator suspends AccessTheWebAsync.
        // - AccessTheWebAsync can't continue until getStringTask is complete.
        // - Meanwhile, control returns to the caller of AccessTheWebAsync.
        // - Control resumes here when getStringTask is complete.
        // - The await operator then retrieves the string result from getStringTask.
        string urlContents = await getStringTask;

        // The return statement specifies an integer result.
        // Any methods that are awaiting AccessTheWebAsync retrieve the length value.
        return urlContents.Length;
    }
}

void DoIndependentWork()
{
    resultsTextBox.Text += "Working . . . . .\r\n";
}
}

// Sample Output:

// Working . . . . .

// Length of the downloaded string: 25035.

```

## See also

- [async](#)
- [await](#)
- [Asynchronous programming](#)
- [Async overview](#)

# Attributes (C#)

1/23/2019 • 5 minutes to read • [Edit Online](#)

Attributes provide a powerful method of associating metadata, or declarative information, with code (assemblies, types, methods, properties, and so forth). After an attribute is associated with a program entity, the attribute can be queried at run time by using a technique called *reflection*. For more information, see [Reflection \(C#\)](#).

Attributes have the following properties:

- Attributes add metadata to your program. *Metadata* is information about the types defined in a program. All .NET assemblies contain a specified set of metadata that describes the types and type members defined in the assembly. You can add custom attributes to specify any additional information that is required. For more information, see, [Creating Custom Attributes \(C#\)](#).
- You can apply one or more attributes to entire assemblies, modules, or smaller program elements such as classes and properties.
- Attributes can accept arguments in the same way as methods and properties.
- Your program can examine its own metadata or the metadata in other programs by using reflection. For more information, see [Accessing Attributes by Using Reflection \(C#\)](#).

## Using attributes

Attributes can be placed on most any declaration, though a specific attribute might restrict the types of declarations on which it is valid. In C#, you specify an attribute by placing the name of the attribute enclosed in square brackets ([ ]) above the declaration of the entity to which it applies.

In this example, the [SerializableAttribute](#) attribute is used to apply a specific characteristic to a class:

```
[Serializable]
public class SampleClass
{
    // Objects of this type can be serialized.
}
```

A method with the attribute [DllImportAttribute](#) is declared like the following example:

```
[System.Runtime.InteropServices.DllImport("user32.dll")]
extern static void SampleMethod();
```

More than one attribute can be placed on a declaration as the following example shows:

```
using System.Runtime.InteropServices;
```

```
void MethodA([In][Out] ref double x) { }
void MethodB([Out][In] ref double x) { }
void MethodC([In, Out] ref double x) { }
```

Some attributes can be specified more than once for a given entity. An example of such a multiuse attribute is [ConditionalAttribute](#):

```
[Conditional("DEBUG"), Conditional("TEST1")]  
void TraceMethod()  
{  
    // ...  
}
```

## NOTE

By convention, all attribute names end with the word "Attribute" to distinguish them from other items in the .NET libraries. However, you do not need to specify the attribute suffix when using attributes in code. For example, `[DllImport]` is equivalent to `[DllImportAttribute]`, but `DllImportAttribute` is the attribute's actual name in the .NET Framework Class Library.

## Attribute parameters

Many attributes have parameters, which can be positional, unnamed, or named. Any positional parameters must be specified in a certain order and cannot be omitted. Named parameters are optional and can be specified in any order. Positional parameters are specified first. For example, these three attributes are equivalent:

```
[DllImport("user32.dll")]  
[DllImport("user32.dll", SetLastError=false, ExactSpelling=false)]  
[DllImport("user32.dll", ExactSpelling=false, SetLastError=false)]
```

The first parameter, the DLL name, is positional and always comes first; the others are named. In this case, both named parameters default to false, so they can be omitted. Positional parameters correspond to the parameters of the attribute constructor. Named or optional parameters correspond to either properties or fields of the attribute. Refer to the individual attribute's documentation for information on default parameter values.

## Attribute targets

The *target* of an attribute is the entity to which the attribute applies. For example, an attribute may apply to a class, a particular method, or an entire assembly. By default, an attribute applies to the element that it precedes. But you can also explicitly identify, for example, whether an attribute is applied to a method, or to its parameter, or to its return value.

To explicitly identify an attribute target, use the following syntax:

```
[target : attribute-list]
```

The list of possible `target` values is shown in the following table.

TARGET VALUE	APPLIES TO
<code>assembly</code>	Entire assembly
<code>module</code>	Current assembly module
<code>field</code>	Field in a class or a struct
<code>event</code>	Event
<code>method</code>	Method or <code>get</code> and <code>set</code> property accessors



TARGET VALUE	APPLIES TO
<code>param</code>	Method parameters or <code>set</code> property accessor parameters
<code>property</code>	Property
<code>return</code>	Return value of a method, property indexer, or <code>get</code> property accessor
<code>type</code>	Struct, class, interface, enum, or delegate

You would specify the `field` target value to apply an attribute to the backing field created for an [auto-implemented property](#).

The following example shows how to apply attributes to assemblies and modules. For more information, see [Common Attributes \(C#\)](#).

```
using System;
using System.Reflection;
[assembly: AssemblyTitle("Production assembly 4")]
[module: CLSCompliant(true)]
```

The following example shows how to apply attributes to methods, method parameters, and method return values in C#.

```
// default: applies to method
[ValidatedContract]
int Method1() { return 0; }

// applies to method
[method: ValidatedContract]
int Method2() { return 0; }

// applies to return value
[return: ValidatedContract]
int Method3() { return 0; }
```

#### NOTE

Regardless of the targets on which `ValidatedContract` is defined to be valid, the `return` target has to be specified, even if `ValidatedContract` were defined to apply only to return values. In other words, the compiler will not use `AttributeUsage` information to resolve ambiguous attribute targets. For more information, see [AttributeUsage \(C#\)](#).

## Common uses for attributes

The following list includes a few of the common uses of attributes in code:

- Marking methods using the `WebMethod` attribute in Web services to indicate that the method should be callable over the SOAP protocol. For more information, see [WebMethodAttribute](#).
- Describing how to marshal method parameters when interoperating with native code. For more information, see [MarshalAsAttribute](#).
- Describing the COM properties for classes, methods, and interfaces.
- Calling unmanaged code using the [DllImportAttribute](#) class.
- Describing your assembly in terms of title, version, description, or trademark.

- Describing which members of a class to serialize for persistence.
- Describing how to map between class members and XML nodes for XML serialization.
- Describing the security requirements for methods.
- Specifying characteristics used to enforce security.
- Controlling optimizations by the just-in-time (JIT) compiler so the code remains easy to debug.
- Obtaining information about the caller to a method.

## Related sections

For more information, see:

- [Creating Custom Attributes \(C#\)](#)
- [Accessing Attributes by Using Reflection \(C#\)](#)
- [How to: Create a C/C++ Union by Using Attributes \(C#\)](#)
- [Common Attributes \(C#\)](#)
- [Caller Information \(C#\)](#)

## See also

- [C# Programming Guide](#)
- [Reflection \(C#\)](#)
- [Attributes](#)
- [Using Attributes in C#](#)

# Caller Information (C#)

10/3/2018 • 2 minutes to read • [Edit Online](#)

By using Caller Info attributes, you can obtain information about the caller to a method. You can obtain file path of the source code, the line number in the source code, and the member name of the caller. This information is helpful for tracing, debugging, and creating diagnostic tools.

To obtain this information, you use attributes that are applied to optional parameters, each of which has a default value. The following table lists the Caller Info attributes that are defined in the [System.Runtime.CompilerServices](#) namespace:

ATTRIBUTE	DESCRIPTION	TYPE
<a href="#">CallerFilePathAttribute</a>	Full path of the source file that contains the caller. This is the file path at compile time.	String
<a href="#">CallerLineNumberAttribute</a>	Line number in the source file at which the method is called.	Integer
<a href="#">CallerMemberNameAttribute</a>	Method or property name of the caller. See <a href="#">Member Names</a> later in this topic.	String

## Example

The following example shows how to use Caller Info attributes. On each call to the `TraceMessage` method, the caller information is substituted as arguments to the optional parameters.

```
public void DoProcessing()
{
    TraceMessage("Something happened.");
}

public void TraceMessage(string message,
    [System.Runtime.CompilerServices.CallerMemberName] string memberName = "",
    [System.Runtime.CompilerServices.CallerFilePath] string sourceFilePath = "",
    [System.Runtime.CompilerServices.CallerLineNumber] int sourceLineNumber = 0)
{
    System.Diagnostics.Trace.WriteLine("message: " + message);
    System.Diagnostics.Trace.WriteLine("member name: " + memberName);
    System.Diagnostics.Trace.WriteLine("source file path: " + sourceFilePath);
    System.Diagnostics.Trace.WriteLine("source line number: " + sourceLineNumber);
}

// Sample Output:
// message: Something happened.
// member name: DoProcessing
// source file path: c:\Visual Studio Projects\CallerInfoCS\CallerInfoCS\Form1.cs
// source line number: 31
```

## Remarks

You must specify an explicit default value for each optional parameter. You can't apply Caller Info attributes to parameters that aren't specified as optional.

The Caller Info attributes don't make a parameter optional. Instead, they affect the default value that's passed in when the argument is omitted.

Caller Info values are emitted as literals into the Intermediate Language (IL) at compile time. Unlike the results of the [StackTrace](#) property for exceptions, the results aren't affected by obfuscation.

You can explicitly supply the optional arguments to control the caller information or to hide caller information.

### Member names

You can use the `CallerMemberName` attribute to avoid specifying the member name as a `String` argument to the called method. By using this technique, you avoid the problem that **Rename Refactoring** doesn't change the `String` values. This benefit is especially useful for the following tasks:

- Using tracing and diagnostic routines.
- Implementing the [INotifyPropertyChanged](#) interface when binding data. This interface allows the property of an object to notify a bound control that the property has changed, so that the control can display the updated information. Without the `CallerMemberName` attribute, you must specify the property name as a literal.

The following chart shows the member names that are returned when you use the `CallerMemberName` attribute.

CALLS OCCURS WITHIN	MEMBER NAME RESULT
Method, property, or event	The name of the method, property, or event from which the call originated.
Constructor	The string ".ctor"
Static constructor	The string ".cctor"
Destructor	The string "Finalize"
User-defined operators or conversions	The generated name for the member, for example, "op_Addition".
Attribute constructor	The name of the method or property to which the attribute is applied. If the attribute is any element within a member (such as a parameter, a return value, or a generic type parameter), this result is the name of the member that's associated with that element.
No containing member (for example, assembly-level or attributes that are applied to types)	The default value of the optional parameter.

### See also

- [Attributes \(C#\)](#)
- [Common Attributes \(C#\)](#)
- [Named and Optional Arguments](#)
- [Programming Concepts \(C#\)](#)

# Collections (C#)

2/3/2019 • 13 minutes to read • [Edit Online](#)

For many applications, you want to create and manage groups of related objects. There are two ways to group objects: by creating arrays of objects, and by creating collections of objects.

Arrays are most useful for creating and working with a fixed number of strongly-typed objects. For information about arrays, see [Arrays](#).

Collections provide a more flexible way to work with groups of objects. Unlike arrays, the group of objects you work with can grow and shrink dynamically as the needs of the application change. For some collections, you can assign a key to any object that you put into the collection so that you can quickly retrieve the object by using the key.

A collection is a class, so you must declare an instance of the class before you can add elements to that collection.

If your collection contains elements of only one data type, you can use one of the classes in the [System.Collections.Generic](#) namespace. A generic collection enforces type safety so that no other data type can be added to it. When you retrieve an element from a generic collection, you do not have to determine its data type or convert it.

## NOTE

For the examples in this topic, include [using](#) directives for the `System.Collections.Generic` and `System.Linq` namespaces.

## In this topic

- [Using a Simple Collection](#)
- [Kinds of Collections](#)
  - [System.Collections.Generic Classes](#)
  - [System.Collections.Concurrent Classes](#)
  - [System.Collections Classes](#)
- [Implementing a Collection of Key/Value Pairs](#)
- [Using LINQ to Access a Collection](#)
- [Sorting a Collection](#)
- [Defining a Custom Collection](#)
- [Iterators](#)

## Using a Simple Collection

The examples in this section use the generic `List<T>` class, which enables you to work with a strongly typed list of objects.

The following example creates a list of strings and then iterates through the strings by using a [foreach](#) statement.

```
// Create a list of strings.
var salmons = new List<string>();
salmons.Add("chinook");
salmons.Add("coho");
salmons.Add("pink");
salmons.Add("sockeye");

// Iterate through the list.
foreach (var salmon in salmons)
{
    Console.Write(salmon + " ");
}
// Output: chinook coho pink sockeye
```

If the contents of a collection are known in advance, you can use a *collection initializer* to initialize the collection. For more information, see [Object and Collection Initializers](#).

The following example is the same as the previous example, except a collection initializer is used to add elements to the collection.

```
// Create a list of strings by using a
// collection initializer.
var salmons = new List<string> { "chinook", "coho", "pink", "sockeye" };

// Iterate through the list.
foreach (var salmon in salmons)
{
    Console.Write(salmon + " ");
}
// Output: chinook coho pink sockeye
```

You can use a [for](#) statement instead of a `foreach` statement to iterate through a collection. You accomplish this by accessing the collection elements by the index position. The index of the elements starts at 0 and ends at the element count minus 1.

The following example iterates through the elements of a collection by using `for` instead of `foreach`.

```
// Create a list of strings by using a
// collection initializer.
var salmons = new List<string> { "chinook", "coho", "pink", "sockeye" };

for (var index = 0; index < salmons.Count; index++)
{
    Console.Write(salmons[index] + " ");
}
// Output: chinook coho pink sockeye
```

The following example removes an element from the collection by specifying the object to remove.

```
// Create a list of strings by using a
// collection initializer.
var salmons = new List<string> { "chinook", "coho", "pink", "sockeye" };

// Remove an element from the list by specifying
// the object.
salmons.Remove("coho");

// Iterate through the list.
foreach (var salmon in salmons)
{
    Console.Write(salmon + " ");
}
// Output: chinook pink sockeye
```

The following example removes elements from a generic list. Instead of a `foreach` statement, a `for` statement that iterates in descending order is used. This is because the `RemoveAt` method causes elements after a removed element to have a lower index value.

```
var numbers = new List<int> { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };

// Remove odd numbers.
for (var index = numbers.Count - 1; index >= 0; index--)
{
    if (numbers[index] % 2 == 1)
    {
        // Remove the element by specifying
        // the zero-based index in the list.
        numbers.RemoveAt(index);
    }
}

// Iterate through the list.
// A lambda expression is placed in the ForEach method
// of the List(T) object.
numbers.ForEach(
    number => Console.Write(number + " "));
// Output: 0 2 4 6 8
```

For the type of elements in the `List<T>`, you can also define your own class. In the following example, the `Galaxy` class that is used by the `List<T>` is defined in the code.

```

private static void IterateThroughList()
{
    var theGalaxies = new List<Galaxy>
    {
        new Galaxy() { Name="Tadpole", MegaLightYears=400},
        new Galaxy() { Name="Pinwheel", MegaLightYears=25},
        new Galaxy() { Name="Milky Way", MegaLightYears=0},
        new Galaxy() { Name="Andromeda", MegaLightYears=3}
    };

    foreach (Galaxy theGalaxy in theGalaxies)
    {
        Console.WriteLine(theGalaxy.Name + " " + theGalaxy.MegaLightYears);
    }

    // Output:
    // Tadpole 400
    // Pinwheel 25
    // Milky Way 0
    // Andromeda 3
}

public class Galaxy
{
    public string Name { get; set; }
    public int MegaLightYears { get; set; }
}

```

## Kinds of Collections

Many common collections are provided by the .NET Framework. Each type of collection is designed for a specific purpose.

Some of the common collection classes are described in this section:

- [System.Collections.Generic](#) classes
- [System.Collections.Concurrent](#) classes
- [System.Collections](#) classes

### System.Collections.Generic Classes

You can create a generic collection by using one of the classes in the [System.Collections.Generic](#) namespace. A generic collection is useful when every item in the collection has the same data type. A generic collection enforces strong typing by allowing only the desired data type to be added.

The following table lists some of the frequently used classes of the [System.Collections.Generic](#) namespace:

CLASS	DESCRIPTION
<a href="#">Dictionary&lt;TKey,TValue&gt;</a>	Represents a collection of key/value pairs that are organized based on the key.
<a href="#">List&lt;T&gt;</a>	Represents a list of objects that can be accessed by index. Provides methods to search, sort, and modify lists.
<a href="#">Queue&lt;T&gt;</a>	Represents a first in, first out (FIFO) collection of objects.
<a href="#">SortedList&lt;TKey,TValue&gt;</a>	Represents a collection of key/value pairs that are sorted by key based on the associated <a href="#">IComparer&lt;T&gt;</a> implementation.



CLASS	DESCRIPTION
<a href="#">Stack&lt;T&gt;</a>	Represents a last in, first out (LIFO) collection of objects.

For additional information, see [Commonly Used Collection Types](#), [Selecting a Collection Class](#), and [System.Collections.Generic](#).

### System.Collections.Concurrent Classes

In the .NET Framework 4 or newer, the collections in the [System.Collections.Concurrent](#) namespace provide efficient thread-safe operations for accessing collection items from multiple threads.

The classes in the [System.Collections.Concurrent](#) namespace should be used instead of the corresponding types in the [System.Collections.Generic](#) and [System.Collections](#) namespaces whenever multiple threads are accessing the collection concurrently. For more information, see [Thread-Safe Collections](#) and [System.Collections.Concurrent](#).

Some classes included in the [System.Collections.Concurrent](#) namespace are [BlockingCollection<T>](#), [ConcurrentDictionary<TKey,TValue>](#), [ConcurrentQueue<T>](#), and [ConcurrentStack<T>](#).

### System.Collections Classes

The classes in the [System.Collections](#) namespace do not store elements as specifically typed objects, but as objects of type `Object`.

Whenever possible, you should use the generic collections in the [System.Collections.Generic](#) namespace or the [System.Collections.Concurrent](#) namespace instead of the legacy types in the `System.Collections` namespace.

The following table lists some of the frequently used classes in the `System.Collections` namespace:

CLASS	DESCRIPTION
<a href="#">ArrayList</a>	Represents an array of objects whose size is dynamically increased as required.
<a href="#">Hashtable</a>	Represents a collection of key/value pairs that are organized based on the hash code of the key.
<a href="#">Queue</a>	Represents a first in, first out (FIFO) collection of objects.
<a href="#">Stack</a>	Represents a last in, first out (LIFO) collection of objects.

The [System.Collections.Specialized](#) namespace provides specialized and strongly typed collection classes, such as string-only collections and linked-list and hybrid dictionaries.

## Implementing a Collection of Key/Value Pairs

The [Dictionary<TKey,TValue>](#) generic collection enables you to access to elements in a collection by using the key of each element. Each addition to the dictionary consists of a value and its associated key. Retrieving a value by using its key is fast because the `Dictionary` class is implemented as a hash table.

The following example creates a `Dictionary` collection and iterates through the dictionary by using a `foreach` statement.

```

private static void IterateThruDictionary()
{
    Dictionary<string, Element> elements = BuildDictionary();

    foreach (KeyValuePair<string, Element> kvp in elements)
    {
        Element theElement = kvp.Value;

        Console.WriteLine("key: " + kvp.Key);
        Console.WriteLine("values: " + theElement.Symbol + " " +
            theElement.Name + " " + theElement.AtomicNumber);
    }
}

private static Dictionary<string, Element> BuildDictionary()
{
    var elements = new Dictionary<string, Element>();

    AddToDictionary(elements, "K", "Potassium", 19);
    AddToDictionary(elements, "Ca", "Calcium", 20);
    AddToDictionary(elements, "Sc", "Scandium", 21);
    AddToDictionary(elements, "Ti", "Titanium", 22);

    return elements;
}

private static void AddToDictionary(Dictionary<string, Element> elements,
    string symbol, string name, int atomicNumber)
{
    Element theElement = new Element();

    theElement.Symbol = symbol;
    theElement.Name = name;
    theElement.AtomicNumber = atomicNumber;

    elements.Add(key: theElement.Symbol, value: theElement);
}

public class Element
{
    public string Symbol { get; set; }
    public string Name { get; set; }
    public int AtomicNumber { get; set; }
}

```

To instead use a collection initializer to build the `Dictionary` collection, you can replace the `BuildDictionary` and `AddToDictionary` methods with the following method.

```

private static Dictionary<string, Element> BuildDictionary2()
{
    return new Dictionary<string, Element>
    {
        {"K",
            new Element() { Symbol="K", Name="Potassium", AtomicNumber=19}},
        {"Ca",
            new Element() { Symbol="Ca", Name="Calcium", AtomicNumber=20}},
        {"Sc",
            new Element() { Symbol="Sc", Name="Scandium", AtomicNumber=21}},
        {"Ti",
            new Element() { Symbol="Ti", Name="Titanium", AtomicNumber=22}}
    };
}

```

The following example uses the `ContainsKey` method and the `Item[TKey]` property of `Dictionary` to quickly find an

item by key. The `Item` property enables you to access an item in the `elements` collection by using the `elements[symbol]` in C#.

```
private static void FindInDictionary(string symbol)
{
    Dictionary<string, Element> elements = BuildDictionary();

    if (elements.ContainsKey(symbol) == false)
    {
        Console.WriteLine(symbol + " not found");
    }
    else
    {
        Element theElement = elements[symbol];
        Console.WriteLine("found: " + theElement.Name);
    }
}
```

The following example instead uses the [TryGetValue](#) method quickly find an item by key.

```
private static void FindInDictionary2(string symbol)
{
    Dictionary<string, Element> elements = BuildDictionary();

    Element theElement = null;
    if (elements.TryGetValue(symbol, out theElement) == false)
        Console.WriteLine(symbol + " not found");
    else
        Console.WriteLine("found: " + theElement.Name);
}
```

## Using LINQ to Access a Collection

LINQ (Language-Integrated Query) can be used to access collections. LINQ queries provide filtering, ordering, and grouping capabilities. For more information, see [Getting Started with LINQ in C#](#).

The following example runs a LINQ query against a generic `List`. The LINQ query returns a different collection that contains the results.

```

private static void ShowLINQ()
{
    List<Element> elements = BuildList();

    // LINQ Query.
    var subset = from theElement in elements
                 where theElement.AtomicNumber < 22
                 orderby theElement.Name
                 select theElement;

    foreach (Element theElement in subset)
    {
        Console.WriteLine(theElement.Name + " " + theElement.AtomicNumber);
    }

    // Output:
    // Calcium 20
    // Potassium 19
    // Scandium 21
}

private static List<Element> BuildList()
{
    return new List<Element>
    {
        { new Element() { Symbol="K", Name="Potassium", AtomicNumber=19}},
        { new Element() { Symbol="Ca", Name="Calcium", AtomicNumber=20}},
        { new Element() { Symbol="Sc", Name="Scandium", AtomicNumber=21}},
        { new Element() { Symbol="Ti", Name="Titanium", AtomicNumber=22}}
    };
}

public class Element
{
    public string Symbol { get; set; }
    public string Name { get; set; }
    public int AtomicNumber { get; set; }
}

```

## Sorting a Collection

The following example illustrates a procedure for sorting a collection. The example sorts instances of the `Car` class that are stored in a `List<T>`. The `Car` class implements the `IComparable<T>` interface, which requires that the `CompareTo` method be implemented.

Each call to the `CompareTo` method makes a single comparison that is used for sorting. User-written code in the `CompareTo` method returns a value for each comparison of the current object with another object. The value returned is less than zero if the current object is less than the other object, greater than zero if the current object is greater than the other object, and zero if they are equal. This enables you to define in code the criteria for greater than, less than, and equal.

In the `ListCars` method, the `cars.Sort()` statement sorts the list. This call to the `Sort` method of the `List<T>` causes the `CompareTo` method to be called automatically for the `Car` objects in the `List`.

```

private static void ListCars()
{
    var cars = new List<Car>
    {
        { new Car() { Name = "car1", Color = "blue", Speed = 20}},
        { new Car() { Name = "car2", Color = "red", Speed = 50}},
        { new Car() { Name = "car3", Color = "green", Speed = 10}},
        { new Car() { Name = "car4", Color = "blue", Speed = 50}},
        { new Car() { Name = "car5", Color = "blue", Speed = 20}}
    };
}

```

```

        { new Car() { Name = "car5", Color = "blue", Speed = 30}},
        { new Car() { Name = "car6", Color = "red", Speed = 60}},
        { new Car() { Name = "car7", Color = "green", Speed = 50}}
    };

    // Sort the cars by color alphabetically, and then by speed
    // in descending order.
    cars.Sort();

    // View all of the cars.
    foreach (Car thisCar in cars)
    {
        Console.WriteLine(thisCar.Color.PadRight(5) + " ");
        Console.WriteLine(thisCar.Speed.ToString() + " ");
        Console.WriteLine(thisCar.Name);
        Console.WriteLine();
    }

    // Output:
    // blue  50 car4
    // blue  30 car5
    // blue  20 car1
    // green 50 car7
    // green 10 car3
    // red   60 car6
    // red   50 car2
}

public class Car : IComparable<Car>
{
    public string Name { get; set; }
    public int Speed { get; set; }
    public string Color { get; set; }

    public int CompareTo(Car other)
    {
        // A call to this method makes a single comparison that is
        // used for sorting.

        // Determine the relative order of the objects being compared.
        // Sort by color alphabetically, and then by speed in
        // descending order.

        // Compare the colors.
        int compare;
        compare = String.Compare(this.Color, other.Color, true);

        // If the colors are the same, compare the speeds.
        if (compare == 0)
        {
            compare = this.Speed.CompareTo(other.Speed);

            // Use descending order for speed.
            compare = -compare;
        }

        return compare;
    }
}

```

## Defining a Custom Collection

You can define a collection by implementing the [IEnumerable<T>](#) or [IEnumerable](#) interface.

Although you can define a custom collection, it is usually better to instead use the collections that are included in the .NET Framework, which are described in [Kinds of Collections](#) earlier in this topic.

The following example defines a custom collection class named `AllColors`. This class implements the `IEnumerable` interface, which requires that the `GetEnumerator` method be implemented.

The `GetEnumerator` method returns an instance of the `ColorEnumerator` class. `ColorEnumerator` implements the `IEnumerator` interface, which requires that the `Current` property, `MoveNext` method, and `Reset` method be implemented.

```
private static void ListColors()
{
    var colors = new AllColors();

    foreach (Color theColor in colors)
    {
        Console.Write(theColor.Name + " ");
    }
    Console.WriteLine();
    // Output: red blue green
}

// Collection class.
public class AllColors : System.Collections.IEnumerable
{
    Color[] _colors =
    {
        new Color() { Name = "red" },
        new Color() { Name = "blue" },
        new Color() { Name = "green" }
    };

    public System.Collections.IEnumerator GetEnumerator()
    {
        return new ColorEnumerator(_colors);

        // Instead of creating a custom enumerator, you could
        // use the GetEnumerator of the array.
        //return _colors.GetEnumerator();
    }

    // Custom enumerator.
    private class ColorEnumerator : System.Collections.IEnumerator
    {
        private Color[] _colors;
        private int _position = -1;

        public ColorEnumerator(Color[] colors)
        {
            _colors = colors;
        }

        object System.Collections.IEnumerator.Current
        {
            get
            {
                return _colors[_position];
            }
        }

        bool System.Collections.IEnumerator.MoveNext()
        {
            _position++;
            return (_position < _colors.Length);
        }

        void System.Collections.IEnumerator.Reset()
        {
            _position = -1;
        }
    }
}
```

```

    }
}

// Element class.
public class Color
{
    public string Name { get; set; }
}

```

## Iterators

An *iterator* is used to perform a custom iteration over a collection. An iterator can be a method or a `get` accessor. An iterator uses a `yield return` statement to return each element of the collection one at a time.

You call an iterator by using a `foreach` statement. Each iteration of the `foreach` loop calls the iterator. When a `yield return` statement is reached in the iterator, an expression is returned, and the current location in code is retained. Execution is restarted from that location the next time that the iterator is called.

For more information, see [Iterators \(C#\)](#).

The following example uses an iterator method. The iterator method has a `yield return` statement that is inside a `for` loop. In the `ListEvenNumbers` method, each iteration of the `foreach` statement body creates a call to the iterator method, which proceeds to the next `yield return` statement.

```

private static void ListEvenNumbers()
{
    foreach (int number in EvenSequence(5, 18))
    {
        Console.Write(number.ToString() + " ");
    }
    Console.WriteLine();
    // Output: 6 8 10 12 14 16 18
}

private static IEnumerable<int> EvenSequence(
    int firstNumber, int lastNumber)
{
    // Yield even numbers in the range.
    for (var number = firstNumber; number <= lastNumber; number++)
    {
        if (number % 2 == 0)
        {
            yield return number;
        }
    }
}

```

## See also

- [Object and Collection Initializers](#)
- [Programming Concepts \(C#\)](#)
- [Option Strict Statement](#)
- [LINQ to Objects \(C#\)](#)
- [Parallel LINQ \(PLINQ\)](#)
- [Collections and Data Structures](#)
- [Selecting a Collection Class](#)
- [Comparisons and Sorts Within Collections](#)

- [When to Use Generic Collections](#)



# Covariance and Contravariance (C#)

5/4/2018 • 3 minutes to read • [Edit Online](#)

In C#, covariance and contravariance enable implicit reference conversion for array types, delegate types, and generic type arguments. Covariance preserves assignment compatibility and contravariance reverses it.

The following code demonstrates the difference between assignment compatibility, covariance, and contravariance.

```
// Assignment compatibility.
string str = "test";
// An object of a more derived type is assigned to an object of a less derived type.
object obj = str;

// Covariance.
IEnumerable<string> strings = new List<string>();
// An object that is instantiated with a more derived type argument
// is assigned to an object instantiated with a less derived type argument.
// Assignment compatibility is preserved.
IEnumerable<object> objects = strings;

// Contravariance.
// Assume that the following method is in the class:
// static void SetObject(object o) { }
Action<object> actObject = SetObject;
// An object that is instantiated with a less derived type argument
// is assigned to an object instantiated with a more derived type argument.
// Assignment compatibility is reversed.
Action<string> actString = actObject;
```

Covariance for arrays enables implicit conversion of an array of a more derived type to an array of a less derived type. But this operation is not type safe, as shown in the following code example.

```
object[] array = new String[10];
// The following statement produces a run-time exception.
// array[0] = 10;
```

Covariance and contravariance support for method groups allows for matching method signatures with delegate types. This enables you to assign to delegates not only methods that have matching signatures, but also methods that return more derived types (covariance) or that accept parameters that have less derived types (contravariance) than that specified by the delegate type. For more information, see [Variance in Delegates \(C#\)](#) and [Using Variance in Delegates \(C#\)](#).

The following code example shows covariance and contravariance support for method groups.

```

static object GetObject() { return null; }
static void SetObject(object obj) { }

static string GetString() { return ""; }
static void SetString(string str) { }

static void Test()
{
    // Covariance. A delegate specifies a return type as object,
    // but you can assign a method that returns a string.
    Func<object> del = GetString;

    // Contravariance. A delegate specifies a parameter type as string,
    // but you can assign a method that takes an object.
    Action<string> del2 = SetObject;
}

```

In .NET Framework 4 or newer C# supports covariance and contravariance in generic interfaces and delegates and allows for implicit conversion of generic type parameters. For more information, see [Variance in Generic Interfaces \(C#\)](#) and [Variance in Delegates \(C#\)](#).

The following code example shows implicit reference conversion for generic interfaces.

```

IEnumerable<String> strings = new List<String>();
IEnumerable<Object> objects = strings;

```

A generic interface or delegate is called *variant* if its generic parameters are declared covariant or contravariant. C# enables you to create your own variant interfaces and delegates. For more information, see [Creating Variant Generic Interfaces \(C#\)](#) and [Variance in Delegates \(C#\)](#).

## Related Topics

TITLE	DESCRIPTION
<a href="#">Variance in Generic Interfaces (C#)</a>	Discusses covariance and contravariance in generic interfaces and provides a list of variant generic interfaces in the .NET Framework.
<a href="#">Creating Variant Generic Interfaces (C#)</a>	Shows how to create custom variant interfaces.
<a href="#">Using Variance in Interfaces for Generic Collections (C#)</a>	Shows how covariance and contravariance support in the <a href="#">IEnumerable&lt;T&gt;</a> and <a href="#">IComparable&lt;T&gt;</a> interfaces can help you reuse code.
<a href="#">Variance in Delegates (C#)</a>	Discusses covariance and contravariance in generic and non-generic delegates and provides a list of variant generic delegates in the .NET Framework.
<a href="#">Using Variance in Delegates (C#)</a>	Shows how to use covariance and contravariance support in non-generic delegates to match method signatures with delegate types.
<a href="#">Using Variance for Func and Action Generic Delegates (C#)</a>	Shows how covariance and contravariance support in the <code>Func</code> and <code>Action</code> delegates can help you reuse code.

# Expression Trees (C#)

1/23/2019 • 4 minutes to read • [Edit Online](#)

Expression trees represent code in a tree-like data structure, where each node is an expression, for example, a method call or a binary operation such as `x < y`.

You can compile and run code represented by expression trees. This enables dynamic modification of executable code, the execution of LINQ queries in various databases, and the creation of dynamic queries. For more information about expression trees in LINQ, see [How to: Use Expression Trees to Build Dynamic Queries \(C#\)](#).

Expression trees are also used in the dynamic language runtime (DLR) to provide interoperability between dynamic languages and the .NET Framework and to enable compiler writers to emit expression trees instead of Microsoft intermediate language (MSIL). For more information about the DLR, see [Dynamic Language Runtime Overview](#).

You can have the C# or Visual Basic compiler create an expression tree for you based on an anonymous lambda expression, or you can create expression trees manually by using the [System.Linq.Expressions](#) namespace.

## Creating Expression Trees from Lambda Expressions

When a lambda expression is assigned to a variable of type [Expression<TDelegate>](#), the compiler emits code to build an expression tree that represents the lambda expression.

The C# compiler can generate expression trees only from expression lambdas (or single-line lambdas). It cannot parse statement lambdas (or multi-line lambdas). For more information about lambda expressions in C#, see [Lambda Expressions](#).

The following code examples demonstrate how to have the C# compiler create an expression tree that represents the lambda expression `num => num < 5`.

```
Expression<Func<int, bool>> lambda = num => num < 5;
```

## Creating Expression Trees by Using the API

To create expression trees by using the API, use the [Expression](#) class. This class contains static factory methods that create expression tree nodes of specific types, for example, [ParameterExpression](#), which represents a variable or parameter, or [MethodCallExpression](#), which represents a method call. [ParameterExpression](#), [MethodCallExpression](#), and the other expression-specific types are also defined in the [System.Linq.Expressions](#) namespace. These types derive from the abstract type [Expression](#).

The following code example demonstrates how to create an expression tree that represents the lambda expression `num => num < 5` by using the API.

```
// Add the following using directive to your code file:
// using System.Linq.Expressions;

// Manually build the expression tree for
// the lambda expression num => num < 5.
ParameterExpression numParam = Expression.Parameter(typeof(int), "num");
ConstantExpression five = Expression.Constant(5, typeof(int));
BinaryExpression numLessThanFive = Expression.LessThan(numParam, five);
Expression<Func<int, bool>> lambda1 =
    Expression.Lambda<Func<int, bool>>(
        numLessThanFive,
        new ParameterExpression[] { numParam });
```

In .NET Framework 4 or later, the expression trees API also supports assignments and control flow expressions such as loops, conditional blocks, and `try-catch` blocks. By using the API, you can create expression trees that are more complex than those that can be created from lambda expressions by the C# compiler. The following example demonstrates how to create an expression tree that calculates the factorial of a number.

```
// Creating a parameter expression.
ParameterExpression value = Expression.Parameter(typeof(int), "value");

// Creating an expression to hold a local variable.
ParameterExpression result = Expression.Parameter(typeof(int), "result");

// Creating a label to jump to from a loop.
LabelTarget label = Expression.Label(typeof(int));

// Creating a method body.
BlockExpression block = Expression.Block(
    // Adding a local variable.
    new[] { result },
    // Assigning a constant to a local variable: result = 1
    Expression.Assign(result, Expression.Constant(1)),
    // Adding a loop.
    Expression.Loop(
        // Adding a conditional block into the loop.
        Expression.IfThenElse(
            // Condition: value > 1
            Expression.GreaterThan(value, Expression.Constant(1)),
            // If true: result *= value --
            Expression.MultiplyAssign(result,
                Expression.PostDecrementAssign(value)),
            // If false, exit the loop and go to the label.
            Expression.Break(label, result)
        ),
        // Label to jump to.
        label
    )
);

// Compile and execute an expression tree.
int factorial = Expression.Lambda<Func<int, int>>(block, value).Compile()(5);

Console.WriteLine(factorial);
// Prints 120.
```

For more information, see [Generating Dynamic Methods with Expression Trees in Visual Studio 2010](#), which also applies to later versions of Visual Studio.

## Parsing Expression Trees

The following code example demonstrates how the expression tree that represents the lambda expression

`num => num < 5` can be decomposed into its parts.

```
// Add the following using directive to your code file:
// using System.Linq.Expressions;

// Create an expression tree.
Expression<Func<int, bool>> exprTree = num => num < 5;

// Decompose the expression tree.
ParameterExpression param = (ParameterExpression)exprTree.Parameters[0];
BinaryExpression operation = (BinaryExpression)exprTree.Body;
ParameterExpression left = (ParameterExpression)operation.Left;
ConstantExpression right = (ConstantExpression)operation.Right;

Console.WriteLine("Decomposed expression: {0} => {1} {2} {3}",
    param.Name, left.Name, operation.NodeType, right.Value);

// This code produces the following output:

// Decomposed expression: num => num LessThan 5
```

## Immutability of Expression Trees

Expression trees should be immutable. This means that if you want to modify an expression tree, you must construct a new expression tree by copying the existing one and replacing nodes in it. You can use an expression tree visitor to traverse the existing expression tree. For more information, see [How to: Modify Expression Trees \(C#\)](#).

## Compiling Expression Trees

The [Expression<TDelegate>](#) type provides the [Compile](#) method that compiles the code represented by an expression tree into an executable delegate.

The following code example demonstrates how to compile an expression tree and run the resulting code.

```
// Creating an expression tree.
Expression<Func<int, bool>> expr = num => num < 5;

// Compiling the expression tree into a delegate.
Func<int, bool> result = expr.Compile();

// Invoking the delegate and writing the result to the console.
Console.WriteLine(result(4));

// Prints True.

// You can also use simplified syntax
// to compile and run an expression tree.
// The following line can replace two previous statements.
Console.WriteLine(expr.Compile()(4));

// Also prints True.
```

For more information, see [How to: Execute Expression Trees \(C#\)](#).

## See also

- [System.Linq.Expressions](#)
- [How to: Execute Expression Trees \(C#\)](#)

- [How to: Modify Expression Trees \(C#\)](#)
- [Lambda Expressions](#)
- [Dynamic Language Runtime Overview](#)
- [Programming Concepts \(C#\)](#)

# Iterators (C#)

1/23/2019 • 7 minutes to read • [Edit Online](#)

An *iterator* can be used to step through collections such as lists and arrays.

An iterator method or `get` accessor performs a custom iteration over a collection. An iterator method uses the `yield return` statement to return each element one at a time. When a `yield return` statement is reached, the current location in code is remembered. Execution is restarted from that location the next time the iterator function is called.

You consume an iterator from client code by using a `foreach` statement or by using a LINQ query.

In the following example, the first iteration of the `foreach` loop causes execution to proceed in the `SomeNumbers` iterator method until the first `yield return` statement is reached. This iteration returns a value of 3, and the current location in the iterator method is retained. On the next iteration of the loop, execution in the iterator method continues from where it left off, again stopping when it reaches a `yield return` statement. This iteration returns a value of 5, and the current location in the iterator method is again retained. The loop completes when the end of the iterator method is reached.

```
static void Main()
{
    foreach (int number in SomeNumbers())
    {
        Console.Write(number.ToString() + " ");
    }
    // Output: 3 5 8
    Console.ReadKey();
}

public static System.Collections.IEnumerable SomeNumbers()
{
    yield return 3;
    yield return 5;
    yield return 8;
}
```

The return type of an iterator method or `get` accessor can be `IEnumerable`, `IEnumerable<T>`, `IEnumerator`, or `IEnumerator<T>`.

You can use a `yield break` statement to end the iteration.

## NOTE

For all examples in this topic except the Simple Iterator example, include `using` directives for the `System.Collections` and `System.Collections.Generic` namespaces.

## Simple Iterator

The following example has a single `yield return` statement that is inside a `for` loop. In `Main`, each iteration of the `foreach` statement body creates a call to the iterator function, which proceeds to the next `yield return` statement.

```

static void Main()
{
    foreach (int number in EvenSequence(5, 18))
    {
        Console.Write(number.ToString() + " ");
    }
    // Output: 6 8 10 12 14 16 18
    Console.ReadKey();
}

public static System.Collections.Generic.IEnumerable<int>
    EvenSequence(int firstNumber, int lastNumber)
{
    // Yield even numbers in the range.
    for (int number = firstNumber; number <= lastNumber; number++)
    {
        if (number % 2 == 0)
        {
            yield return number;
        }
    }
}

```

## Creating a Collection Class

In the following example, the `DaysOfTheWeek` class implements the `IEnumerable` interface, which requires a `GetEnumerator` method. The compiler implicitly calls the `GetEnumerator` method, which returns an `IEnumerator`.

The `GetEnumerator` method returns each string one at a time by using the `yield return` statement.

```

static void Main()
{
    DaysOfTheWeek days = new DaysOfTheWeek();

    foreach (string day in days)
    {
        Console.Write(day + " ");
    }
    // Output: Sun Mon Tue Wed Thu Fri Sat
    Console.ReadKey();
}

public class DaysOfTheWeek : IEnumerable
{
    private string[] days = { "Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat" };

    public IEnumerator GetEnumerator()
    {
        for (int index = 0; index < days.Length; index++)
        {
            // Yield each day of the week.
            yield return days[index];
        }
    }
}

```

The following example creates a `Zoo` class that contains a collection of animals.

The `foreach` statement that refers to the class instance (`theZoo`) implicitly calls the `GetEnumerator` method. The `foreach` statements that refer to the `Birds` and `Mammals` properties use the `AnimalsForType` named iterator method.



```

static void Main()
{
    Zoo theZoo = new Zoo();

    theZoo.AddMammal("Whale");
    theZoo.AddMammal("Rhinoceros");
    theZoo.AddBird("Penguin");
    theZoo.AddBird("Warbler");

    foreach (string name in theZoo)
    {
        Console.Write(name + " ");
    }
    Console.WriteLine();
    // Output: Whale Rhinoceros Penguin Warbler

    foreach (string name in theZoo.Birds)
    {
        Console.Write(name + " ");
    }
    Console.WriteLine();
    // Output: Penguin Warbler

    foreach (string name in theZoo.Mammals)
    {
        Console.Write(name + " ");
    }
    Console.WriteLine();
    // Output: Whale Rhinoceros

    Console.ReadKey();
}

public class Zoo : IEnumerable
{
    // Private members.
    private List<Animal> animals = new List<Animal>();

    // Public methods.
    public void AddMammal(string name)
    {
        animals.Add(new Animal { Name = name, Type = Animal.TypeEnum.Mammal });
    }

    public void AddBird(string name)
    {
        animals.Add(new Animal { Name = name, Type = Animal.TypeEnum.Bird });
    }

    public IEnumerator GetEnumerator()
    {
        foreach (Animal theAnimal in animals)
        {
            yield return theAnimal.Name;
        }
    }

    // Public members.
    public IEnumerable Mammals
    {
        get { return AnimalsForType(Animal.TypeEnum.Mammal); }
    }

    public IEnumerable Birds
    {
        get { return AnimalsForType(Animal.TypeEnum.Bird); }
    }

    // Private methods.

```

```
// Private method.
private IEnumerable AnimalsForType(Animal.TypeEnum type)
{
    foreach (Animal theAnimal in animals)
    {
        if (theAnimal.Type == type)
        {
            yield return theAnimal.Name;
        }
    }
}

// Private class.
private class Animal
{
    public enum TypeEnum { Bird, Mammal }

    public string Name { get; set; }
    public TypeEnum Type { get; set; }
}
}
```

## Using Iterators with a Generic List

In the following example, the `Stack<T>` generic class implements the `IEnumerable<T>` generic interface. The `Push` method assigns values to an array of type `T`. The `GetEnumerator` method returns the array values by using the `yield return` statement.

In addition to the generic `GetEnumerator` method, the non-generic `GetEnumerator` method must also be implemented. This is because `IEnumerable<T>` inherits from `IEnumerable`. The non-generic implementation defers to the generic implementation.

The example uses named iterators to support various ways of iterating through the same collection of data. These named iterators are the `TopToBottom` and `BottomToTop` properties, and the `TopN` method.

The `BottomToTop` property uses an iterator in a `get` accessor.

```
static void Main()
{
    Stack<int> theStack = new Stack<int>();

    // Add items to the stack.
    for (int number = 0; number <= 9; number++)
    {
        theStack.Push(number);
    }

    // Retrieve items from the stack.
    // foreach is allowed because theStack implements IEnumerable<int>.
    foreach (int number in theStack)
    {
        Console.Write("{0} ", number);
    }
    Console.WriteLine();
    // Output: 9 8 7 6 5 4 3 2 1 0

    // foreach is allowed, because theStack.TopToBottom returns IEnumerable(Of Integer).
    foreach (int number in theStack.TopToBottom)
    {
        Console.Write("{0} ", number);
    }
    Console.WriteLine();
    // Output: 9 8 7 6 5 4 3 2 1 0

    foreach (int number in theStack.BottomToTop)
```

```

        foreach (int number in theStack.BottomToTop)
        {
            Console.WriteLine("{0} ", number);
        }
        Console.WriteLine();
        // Output: 0 1 2 3 4 5 6 7 8 9

        foreach (int number in theStack.TopN(7))
        {
            Console.WriteLine("{0} ", number);
        }
        Console.WriteLine();
        // Output: 9 8 7 6 5 4 3

        Console.ReadKey();
    }
}

public class Stack<T> : IEnumerable<T>
{
    private T[] values = new T[100];
    private int top = 0;

    public void Push(T t)
    {
        values[top] = t;
        top++;
    }
    public T Pop()
    {
        top--;
        return values[top];
    }

    // This method implements the GetEnumerator method. It allows
    // an instance of the class to be used in a foreach statement.
    public IEnumerator<T> GetEnumerator()
    {
        for (int index = top - 1; index >= 0; index--)
        {
            yield return values[index];
        }
    }

    IEnumerator IEnumerable.GetEnumerator()
    {
        return GetEnumerator();
    }

    public IEnumerable<T> TopToBottom
    {
        get { return this; }
    }

    public IEnumerable<T> BottomToTop
    {
        get
        {
            for (int index = 0; index <= top - 1; index++)
            {
                yield return values[index];
            }
        }
    }

    public IEnumerable<T> TopN(int itemsFromTop)
    {
        // Return less than itemsFromTop if necessary.
        int startIndex = itemsFromTop >= top ? 0 : top - itemsFromTop;
        for (int index = top - 1; index >= startIndex; index--)

```

```

        for (int index = top - 1; index >= start index; index--)
        {
            yield return values[index];
        }
    }
}

```

## Syntax Information

An iterator can occur as a method or `get` accessor. An iterator cannot occur in an event, instance constructor, static constructor, or static finalizer.

An implicit conversion must exist from the expression type in the `yield return` statement to the type argument for the `IEnumerator` returned by the iterator.

In C#, an iterator method cannot have any `in`, `ref`, or `out` parameters.

In C#, "yield" is not a reserved word and has special meaning only when it is used before a `return` or `break` keyword.

## Technical Implementation

Although you write an iterator as a method, the compiler translates it into a nested class that is, in effect, a state machine. This class keeps track of the position of the iterator as long the `foreach` loop in the client code continues.

To see what the compiler does, you can use the `Ildasm.exe` tool to view the Microsoft intermediate language code that's generated for an iterator method.

When you create an iterator for a `class` or `struct`, you don't have to implement the whole `IEnumerator` interface. When the compiler detects the iterator, it automatically generates the `Current`, `MoveNext`, and `Dispose` methods of the `IEnumerator` or `IEnumerator<T>` interface.

On each successive iteration of the `foreach` loop (or the direct call to `IEnumerator.MoveNext`), the next iterator code body resumes after the previous `yield return` statement. It then continues to the next `yield return` statement until the end of the iterator body is reached, or until a `yield break` statement is encountered.

Iterators don't support the `IEnumerator.Reset` method. To reiterate from the start, you must obtain a new iterator. Calling `Reset` on the iterator returned by an iterator method throws a `NotSupportedException`.

For additional information, see the [C# Language Specification](#).

## Use of Iterators

Iterators enable you to maintain the simplicity of a `foreach` loop when you need to use complex code to populate a list sequence. This can be useful when you want to do the following:

- Modify the list sequence after the first `foreach` loop iteration.
- Avoid fully loading a large list before the first iteration of a `foreach` loop. An example is a paged fetch to load a batch of table rows. Another example is the `EnumerateFiles` method, which implements iterators within the .NET Framework.
- Encapsulate building the list in the iterator. In the iterator method, you can build the list and then yield each result in a loop.

## See also

- `System.Collections.Generic`
- `IEnumerable<T>`
- `foreach`, `in`
- `yield`
- Using `foreach` with Arrays
- Generics

# Language Integrated Query (LINQ)

10/13/2018 • 3 minutes to read • [Edit Online](#)

Language-Integrated Query (LINQ) is the name for a set of technologies based on the integration of query capabilities directly into the C# language. Traditionally, queries against data are expressed as simple strings without type checking at compile time or IntelliSense support. Furthermore, you have to learn a different query language for each type of data source: SQL databases, XML documents, various Web services, and so on. With LINQ, a query is a first-class language construct, just like classes, methods, events.

For a developer who writes queries, the most visible "language-integrated" part of LINQ is the query expression. Query expressions are written in a declarative *query syntax*. By using query syntax, you can perform filtering, ordering, and grouping operations on data sources with a minimum of code. You use the same basic query expression patterns to query and transform data in SQL databases, ADO .NET Datasets, XML documents and streams, and .NET collections.

The following example shows the complete query operation. The complete operation includes creating a data source, defining the query expression, and executing the query in a `foreach` statement.

```
class LINQQueryExpressions
{
    static void Main()
    {
        // Specify the data source.
        int[] scores = new int[] { 97, 92, 81, 60 };

        // Define the query expression.
        IEnumerable<int> scoreQuery =
            from score in scores
            where score > 80
            select score;

        // Execute the query.
        foreach (int i in scoreQuery)
        {
            Console.Write(i + " ");
        }
    }
}
// Output: 97 92 81
```

## Query expression overview

- Query expressions can be used to query and to transform data from any LINQ-enabled data source. For example, a single query can retrieve data from a SQL database, and produce an XML stream as output.
- Query expressions are easy to master because they use many familiar C# language constructs.
- The variables in a query expression are all strongly typed, although in many cases you do not have to provide the type explicitly because the compiler can infer it. For more information, see [Type relationships in LINQ query operations](#).
- A query is not executed until you iterate over the query variable, for example, in a `foreach` statement. For more information, see [Introduction to LINQ queries](#).

- At compile time, query expressions are converted to Standard Query Operator method calls according to the rules set forth in the C# specification. Any query that can be expressed by using query syntax can also be expressed by using method syntax. However, in most cases query syntax is more readable and concise. For more information, see [C# language specification](#) and [Standard query operators overview](#).
- As a rule when you write LINQ queries, we recommend that you use query syntax whenever possible and method syntax whenever necessary. There is no semantic or performance difference between the two different forms. Query expressions are often more readable than equivalent expressions written in method syntax.
- Some query operations, such as [Count](#) or [Max](#), have no equivalent query expression clause and must therefore be expressed as a method call. Method syntax can be combined with query syntax in various ways. For more information, see [Query syntax and method syntax in LINQ](#).
- Query expressions can be compiled to expression trees or to delegates, depending on the type that the query is applied to. [IEnumerable<T>](#) queries are compiled to delegates. [IQueryable](#) and [IQueryable<T>](#) queries are compiled to expression trees. For more information, see [Expression trees](#).

## Next steps

To learn more details about LINQ, start by becoming familiar with some basic concepts in [Query expression basics](#), and then read the documentation for the LINQ technology in which you are interested:

- XML documents: [LINQ to XML](#)
- ADO.NET Entity Framework: [LINQ to entities](#)
- .NET collections, files, strings and so on: [LINQ to objects](#)

To gain a deeper understanding of LINQ in general, see [LINQ in C#](#).

To start working with LINQ in C#, see the tutorial [Working with LINQ](#).

# Object-Oriented Programming (C#)

1/23/2019 • 9 minutes to read • [Edit Online](#)

C# provides full support for object-oriented programming including encapsulation, inheritance, and polymorphism.

*Encapsulation* means that a group of related properties, methods, and other members are treated as a single unit or object.

*Inheritance* describes the ability to create new classes based on an existing class.

*Polymorphism* means that you can have multiple classes that can be used interchangeably, even though each class implements the same properties or methods in different ways.

This section describes the following concepts:

- [Classes and Objects](#)
  - [Class Members](#)
    - [Properties and Fields](#)
    - [Methods](#)
    - [Constructors](#)
    - [Finalizers](#)
    - [Events](#)
    - [Nested Classes](#)
  - [Access Modifiers and Access Levels](#)
  - [Instantiating Classes](#)
  - [Static Classes and Members](#)
  - [Anonymous Types](#)
- [Inheritance](#)
  - [Overriding Members](#)
- [Interfaces](#)
- [Generics](#)
- [Delegates](#)

## Classes and Objects

The terms *class* and *object* are sometimes used interchangeably, but in fact, classes describe the *type* of objects, while objects are usable *instances* of classes. So, the act of creating an object is called *instantiation*. Using the blueprint analogy, a class is a blueprint, and an object is a building made from that blueprint.

To define a class:



```
class SampleClass
{
}
```

C# also provides a light version of classes called *structures* that are useful when you need to create large array of objects and do not want to consume too much memory for that.

To define a structure:

```
struct SampleStruct
{
}
```

For more information, see:

- [class](#)
- [struct](#)

## Class Members

Each class can have different *class members* that include properties that describe class data, methods that define class behavior, and events that provide communication between different classes and objects.

### Properties and Fields

Fields and properties represent information that an object contains. Fields are like variables because they can be read or set directly.

To define a field:

```
class SampleClass
{
    public string sampleField;
}
```

Properties have get and set procedures, which provide more control on how values are set or returned.

C# allows you either to create a private field for storing the property value or use so-called auto-implemented properties that create this field automatically behind the scenes and provide the basic logic for the property procedures.

To define an auto-implemented property:

```
class SampleClass
{
    public int SampleProperty { get; set; }
}
```

If you need to perform some additional operations for reading and writing the property value, define a field for storing the property value and provide the basic logic for storing and retrieving it:

```

class SampleClass
{
    private int _sample;
    public int Sample
    {
        // Return the value stored in a field.
        get { return _sample; }
        // Store the value in the field.
        set { _sample = value; }
    }
}

```

Most properties have methods or procedures to both set and get the property value. However, you can create read-only or write-only properties to restrict them from being modified or read. In C#, you can omit the `get` or `set` property method. However, auto-implemented properties cannot be read-only or write-only.

For more information, see:

- [get](#)
- [set](#)

### Methods

A *method* is an action that an object can perform.

To define a method of a class:

```

class SampleClass
{
    public int sampleMethod(string sampleParam)
    {
        // Insert code here
    }
}

```

A class can have several implementations, or *overloads*, of the same method that differ in the number of parameters or parameter types.

To overload a method:

```

public int sampleMethod(string sampleParam) {};
public int sampleMethod(int sampleParam) {}

```

In most cases you declare a method within a class definition. However, C# also supports *extension methods* that allow you to add methods to an existing class outside the actual definition of the class.

For more information, see:

- [Methods](#)
- [Extension Methods](#)

### Constructors

Constructors are class methods that are executed automatically when an object of a given type is created. Constructors usually initialize the data members of the new object. A constructor can run only once when a class is created. Furthermore, the code in the constructor always runs before any other code in a class. However, you can create multiple constructor overloads in the same way as for any other method.

To define a constructor for a class:

```
public class SampleClass
{
    public SampleClass()
    {
        // Add code here
    }
}
```

For more information, see:

[Constructors.](#)

### Finalizers

Finalizers are used to destruct instances of classes. In the .NET Framework, the garbage collector automatically manages the allocation and release of memory for the managed objects in your application. However, you may still need finalizers to clean up any unmanaged resources that your application creates. There can be only one finalizers for a class.

For more information about finalizers and garbage collection in the .NET Framework, see [Garbage Collection](#).

### Events

Events enable a class or object to notify other classes or objects when something of interest occurs. The class that sends (or raises) the event is called the *publisher* and the classes that receive (or handle) the event are called *subscribers*. For more information about events, how they are raised and handled, see [Events](#).

- To declare an event in a class, use the [event](#) keyword.
- To raise an event, invoke the event delegate.
- To subscribe to an event, use the `+=` operator; to unsubscribe from an event, use the `-=` operator.

### Nested Classes

A class defined within another class is called *nested*. By default, the nested class is private.

```
class Container
{
    class Nested
    {
        // Add code here.
    }
}
```

To create an instance of the nested class, use the name of the container class followed by the dot and then followed by the name of the nested class:

```
Container.Nested nestedInstance = new Container.Nested()
```

### Access Modifiers and Access Levels

All classes and class members can specify what access level they provide to other classes by using *access modifiers*.

The following access modifiers are available:

C# MODIFIER	DEFINITION
<a href="#">public</a>	The type or member can be accessed by any other code in the same assembly or another assembly that references it.

C# MODIFIER	DEFINITION
<code>private</code>	The type or member can only be accessed by code in the same class.
<code>protected</code>	The type or member can only be accessed by code in the same class or in a derived class.
<code>internal</code>	The type or member can be accessed by any code in the same assembly, but not from another assembly.
<code>protected internal</code>	The type or member can be accessed by any code in the same assembly, or by any derived class in another assembly.
<code>private protected</code>	The type or member can be accessed by code in the same class or in a derived class within the base class assembly.

For more information, see [Access Modifiers](#).

## Instantiating Classes

To create an object, you need to instantiate a class, or create a class instance.

```
SampleClass sampleObject = new SampleClass();
```

After instantiating a class, you can assign values to the instance's properties and fields and invoke class methods.

```
// Set a property value.
sampleObject.sampleProperty = "Sample String";
// Call a method.
sampleObject.sampleMethod();
```

To assign values to properties during the class instantiation process, use object initializers:

```
// Set a property value.
SampleClass sampleObject = new SampleClass
{ FirstProperty = "A", SecondProperty = "B" };
```

For more information, see:

- [new Operator](#)
- [Object and Collection Initializers](#)

## Static Classes and Members

A static member of the class is a property, procedure, or field that is shared by all instances of a class.

To define a static member:

```
static class SampleClass
{
    public static string SampleString = "Sample String";
}
```

To access the static member, use the name of the class without creating an object of this class:

```
Console.WriteLine(SampleClass.SampleString);
```

Static classes in C# have static members only and cannot be instantiated. Static members also cannot access non-static properties, fields or methods

For more information, see: [static](#).

### Anonymous Types

Anonymous types enable you to create objects without writing a class definition for the data type. Instead, the compiler generates a class for you. The class has no usable name and contains the properties you specify in declaring the object.

To create an instance of an anonymous type:

```
// sampleObject is an instance of a simple anonymous type.
var sampleObject =
    new { FirstProperty = "A", SecondProperty = "B" };
```

For more information, see: [Anonymous Types](#).

## Inheritance

Inheritance enables you to create a new class that reuses, extends, and modifies the behavior that is defined in another class. The class whose members are inherited is called the *base class*, and the class that inherits those members is called the *derived class*. However, all classes in C# implicitly inherit from the [Object](#) class that supports .NET class hierarchy and provides low-level services to all classes.

#### NOTE

C# doesn't support multiple inheritance. That is, you can specify only one base class for a derived class.

To inherit from a base class:

```
class DerivedClass:BaseClass { }
```

By default all classes can be inherited. However, you can specify whether a class must not be used as a base class, or create a class that can be used as a base class only.

To specify that a class cannot be used as a base class:

```
public sealed class A { }
```

To specify that a class can be used as a base class only and cannot be instantiated:

```
public abstract class B { }
```

For more information, see:

- [sealed](#)
- [abstract](#)

### Overriding Members

By default, a derived class inherits all members from its base class. If you want to change the behavior of the inherited member, you need to override it. That is, you can define a new implementation of the method, property or event in the derived class.

The following modifiers are used to control how properties and methods are overridden:

C# MODIFIER	DEFINITION
<a href="#">virtual</a>	Allows a class member to be overridden in a derived class.
<a href="#">override</a>	Overrides a virtual (overridable) member defined in the base class.
<a href="#">abstract</a>	Requires that a class member to be overridden in the derived class.
<a href="#">new Modifier</a>	Hides a member inherited from a base class

## Interfaces

Interfaces, like classes, define a set of properties, methods, and events. But unlike classes, interfaces do not provide implementation. They are implemented by classes, and defined as separate entities from classes. An interface represents a contract, in that a class that implements an interface must implement every aspect of that interface exactly as it is defined.

To define an interface:

```
interface ISampleInterface
{
    void doSomething();
}
```

To implement an interface in a class:

```
class SampleClass : ISampleInterface
{
    void ISampleInterface.doSomething()
    {
        // Method implementation.
    }
}
```

For more information, see:

[Interfaces](#)

[interface](#)

## Generics

Classes, structures, interfaces and methods in the .NET Framework can include *type parameters* that define types of objects that they can store or use. The most common example of generics is a collection, where you can specify the type of objects to be stored in a collection.

To define a generic class:

```
public class SampleGeneric<T>
{
    public T Field;
}
```

To create an instance of a generic class:

```
SampleGeneric<string> sampleObject = new SampleGeneric<string>();
sampleObject.Field = "Sample string";
```

For more information, see:

- [Generics](#)
- [Generics](#)

## Delegates

A *delegate* is a type that defines a method signature, and can provide a reference to any method with a compatible signature. You can invoke (or call) the method through the delegate. Delegates are used to pass methods as arguments to other methods.

### NOTE

Event handlers are nothing more than methods that are invoked through delegates. For more information about using delegates in event handling, see [Events](#).

To create a delegate:

```
public delegate void SampleDelegate(string str);
```

To create a reference to a method that matches the signature specified by the delegate:

```
class SampleClass
{
    // Method that matches the SampleDelegate signature.
    public static void sampleMethod(string message)
    {
        // Add code here.
    }
    // Method that instantiates the delegate.
    void SampleDelegate()
    {
        SampleDelegate sd = sampleMethod;
        sd("Sample string");
    }
}
```

For more information, see:

- [Delegates](#)
- [delegate](#)

## See also

- [C# Programming Guide](#)



# Reflection (C#)

1/23/2019 • 2 minutes to read • [Edit Online](#)

Reflection provides objects (of type `Type`) that describe assemblies, modules and types. You can use reflection to 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. If you are using attributes in your code, reflection enables you to access them. For more information, see [Attributes](#).

Here's a simple example of reflection using the static method `GetType` - inherited by all types from the `Object` base class - to obtain the type of a variable:

```
// Using GetType to obtain type information:  
int i = 42;  
System.Type type = i.GetType();  
System.Console.WriteLine(type);
```

The output is:

```
System.Int32
```

The following example uses reflection to obtain the full name of the loaded assembly.

```
// Using Reflection to get information from an Assembly:  
System.Reflection.Assembly info = typeof(System.Int32).Assembly;  
System.Console.WriteLine(info);
```

The output is:

```
mscorlib, Version=2.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089
```

## NOTE

The C# keywords `protected` and `internal` have no meaning in IL and are not used in the reflection APIs. The corresponding terms in IL are *Family* and *Assembly*. To identify an `internal` method using reflection, use the `IsAssembly` property. To identify a `protected internal` method, use the `IsFamilyOrAssembly`.

## Reflection Overview

Reflection is useful in the following situations:

- When you have to access attributes in your program's metadata. For more information, see [Retrieving Information Stored in Attributes](#).
- For examining and instantiating types in an assembly.
- For building new types at runtime. Use classes in [System.Reflection.Emit](#).
- For performing late binding, accessing methods on types created at run time. See the topic [Dynamically Loading and Using Types](#).

## Related Sections

For more information:

- [Reflection](#)
- [Viewing Type Information](#)
- [Reflection and Generic Types](#)
- [System.Reflection.Emit](#)
- [Retrieving Information Stored in Attributes](#)

## See also

- [C# Programming Guide](#)
- [Assemblies in the Common Language Runtime](#)

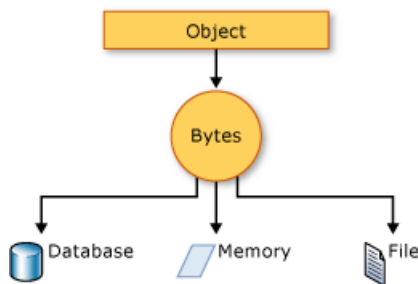
# Serialization (C#)

9/5/2018 • 3 minutes to read • [Edit Online](#)

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 in order to be able to recreate it when needed. The reverse process is called deserialization.

## How serialization works

This illustration shows the overall process of serialization.



The object is serialized to a stream, which carries not just the data, but information about the object's type, such as its version, culture, and assembly name. From that stream, it can be stored in a database, a file, or memory.

### Uses for serialization

Serialization allows the developer to save the state of an object and recreate it as needed, providing storage of objects as well as data exchange. Through serialization, a developer can perform actions like sending the object to a remote application by means of a Web Service, passing an object from one domain to another, passing an object through a firewall as an XML string, or maintaining security or user-specific information across applications.

### Making an object serializable

To serialize an object, you need the object to be serialized, a stream to contain the serialized object, and a [Formatter](#). [System.Runtime.Serialization](#) contains the classes necessary for serializing and deserializing objects.

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

If you don't want a field within your class to be serializable, 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.

## Binary and XML serialization

You can use binary or XML serialization. In binary serialization, all members, even members that are read-only, are serialized, and performance is enhanced. XML serialization provides more readable code, and greater flexibility of object sharing and usage for interoperability purposes.

### Binary serialization

Binary serialization uses binary encoding to produce compact serialization for uses such as storage or socket-

based network streams.

### XML serialization

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 the classes necessary 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.

## Basic and custom serialization

Serialization can be performed in two ways, basic and custom. Basic serialization uses the .NET Framework to automatically serialize the object.

### Basic serialization

The only requirement in basic serialization is that the object 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.

### Custom serialization

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, you must use a custom constructor.

## Designer serialization

Designer serialization is a special form of serialization that involves the kind of object persistence associated with development tools. Designer serialization is the process of converting an object graph into a source file that can later be used to recover the object graph. A source file can contain code, markup, or even SQL table information.

## Related Topics and Examples

[Walkthrough: Persisting an Object in Visual Studio \(C#\)](#)

Demonstrates how serialization can be used to persist an object's data between instances, allowing you to store values and retrieve them the next time the object is instantiated.

[How to: Read Object Data from an XML File \(C#\)](#)

Shows how to read object data that was previously written to an XML file using the [XmlSerializer](#) class.

[How to: Write Object Data to an XML File \(C#\)](#)

Shows how to write the object from a class to an XML file using the [XmlSerializer](#) class.